Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Fall & Winter 2025
Lecture 7

Greedy Algorithms

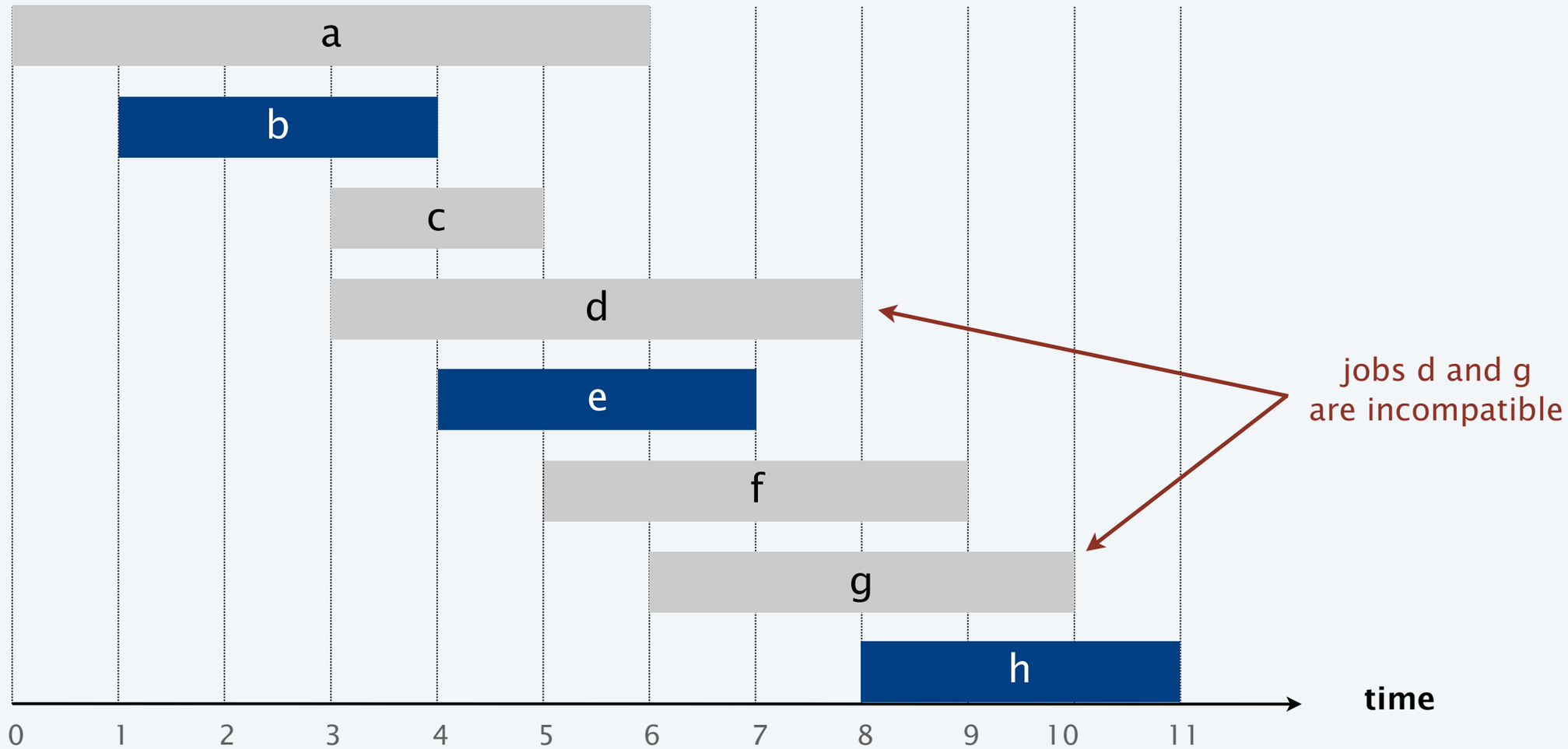
- Active selection (aka interval scheduling)
- Huffman coding
- Minimum spanning trees
- Take-home messages

Greedy Algorithms

- Active selection (aka interval scheduling)
- Huffman coding
- Minimum spanning trees
- Take-home messages

Interval scheduling

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Greedy algorithms I: quiz 2

Consider jobs in some order, taking each job provided it's compatible with the ones already taken. Which rule is optimal?

A. [Earliest start time] Consider jobs in ascending order of s_j.

B. [Earliest finish time] Consider jobs in ascending order of f_j.

C. [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

D. None of the above.

Interval scheduling: earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

$S \leftarrow \emptyset$. ← set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

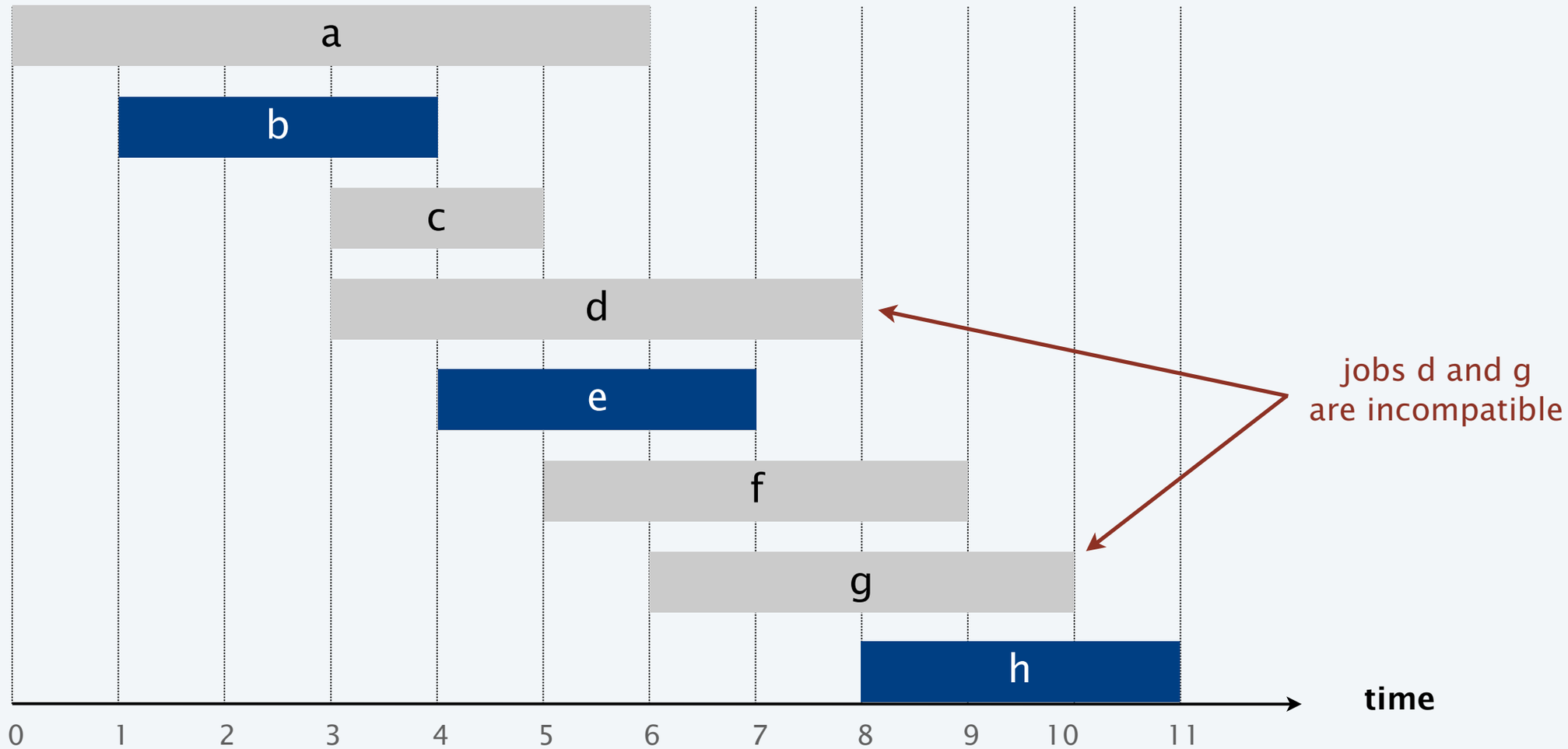
RETURN S.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

- Keep track of job j^* that was added last to S.
- Job j is compatible with S iff $s_j \geq f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.

Interval scheduling

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

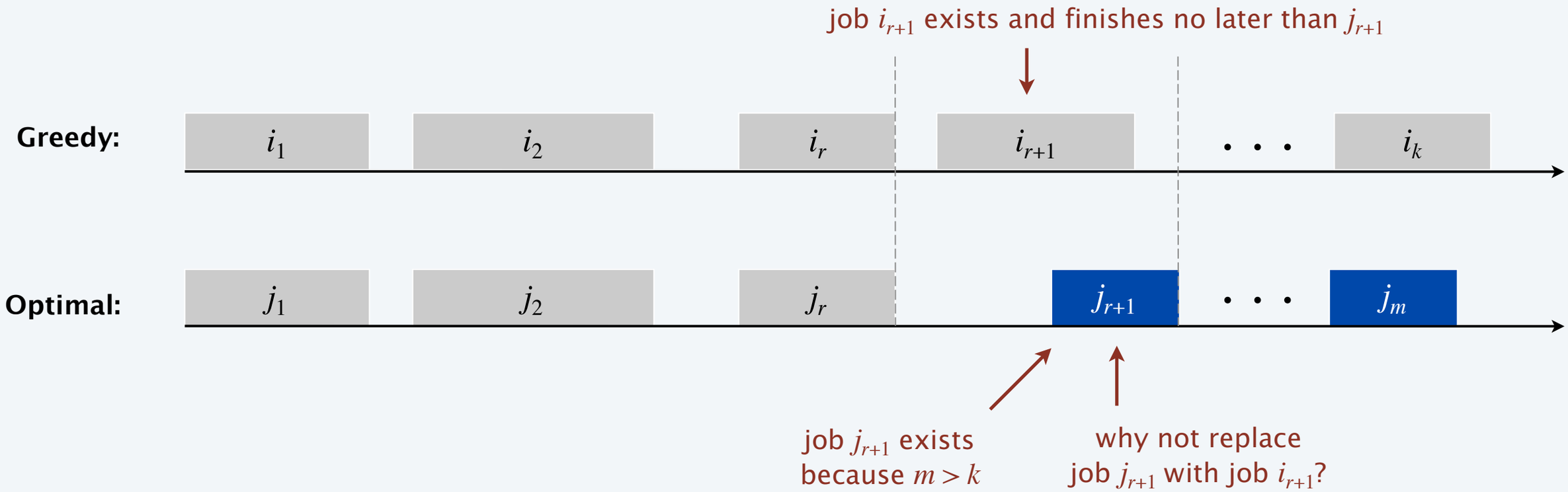


Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

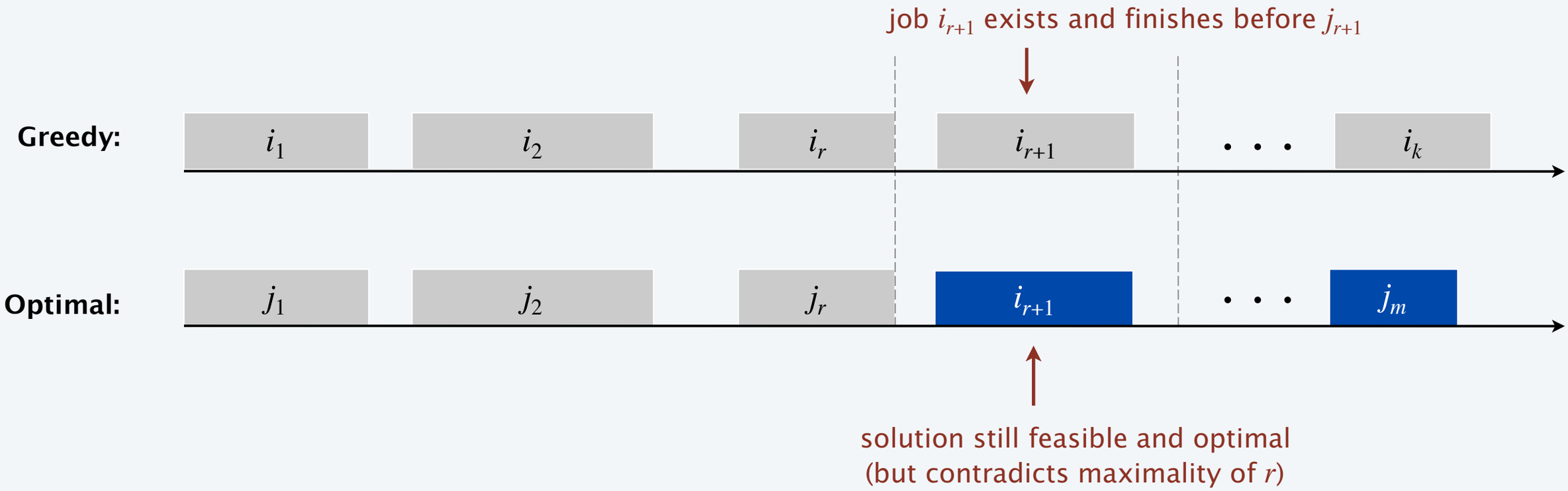


Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Greedy algorithms I: quiz 3

Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals. Is the earliest-finish-time-first algorithm still optimal?

A. Yes, because greedy algorithms are always optimal.

B. Yes, because the same proof of correctness is valid.

C. No, because the same proof of correctness is no longer valid.

D. No, because you could assign a huge weight to a job that overlaps the job with the earliest finish time.

❖ Optimization Problems:

Given a set of constraints and an optimization function. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called an optimal solution.

❖ Optimization Problems:

Given a set of constraints and an optimization function. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called an optimal solution.

❖ The Greedy Method:

Make the best decision at each stage, under some greedy criterion. A decision made in one stage is not changed in a later stage, so each decision should assure feasibility.

Note:

- **Greedy algorithm works only if the local optimum is equal to the global optimum.**
- **Greedy algorithm does not guarantee optimal solutions. However, it generally produces solutions that are very close in value (heuristics) to the optimal, and hence is intuitively appealing when finding the optimal solution takes too much time.**

Activity Selection Problem

Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

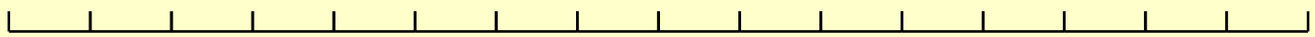
Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

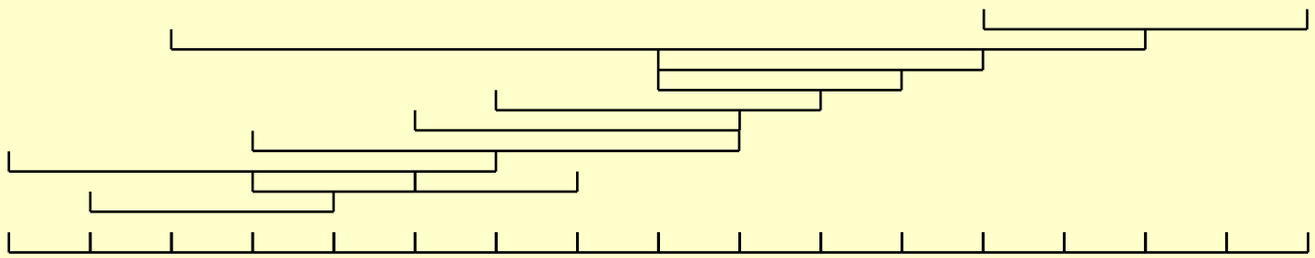
Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

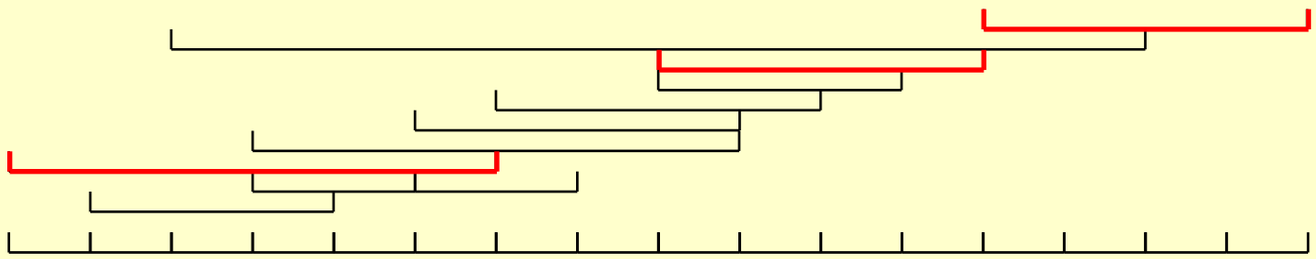
Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

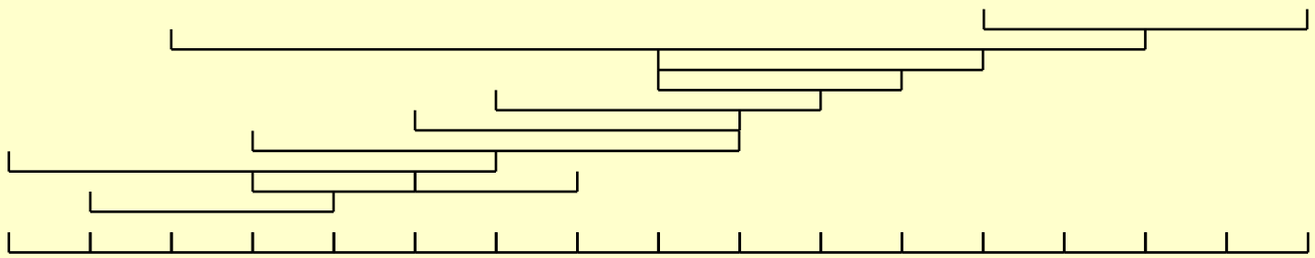
Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

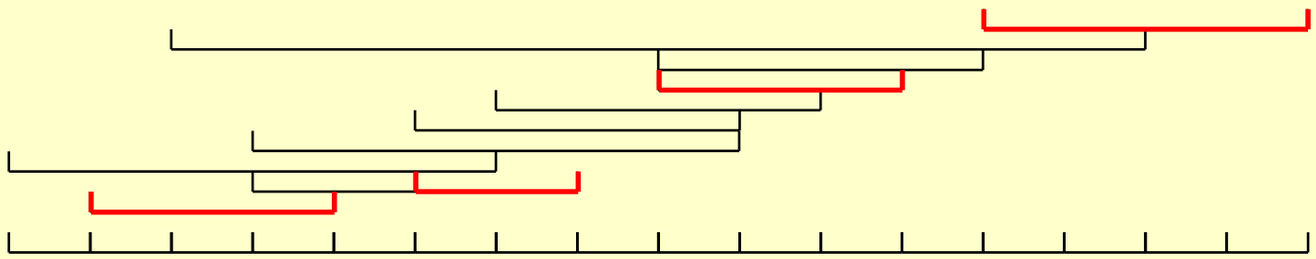
Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

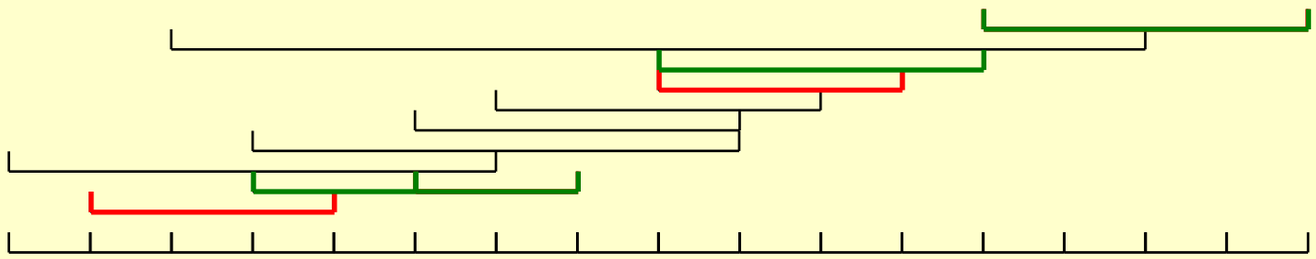
Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

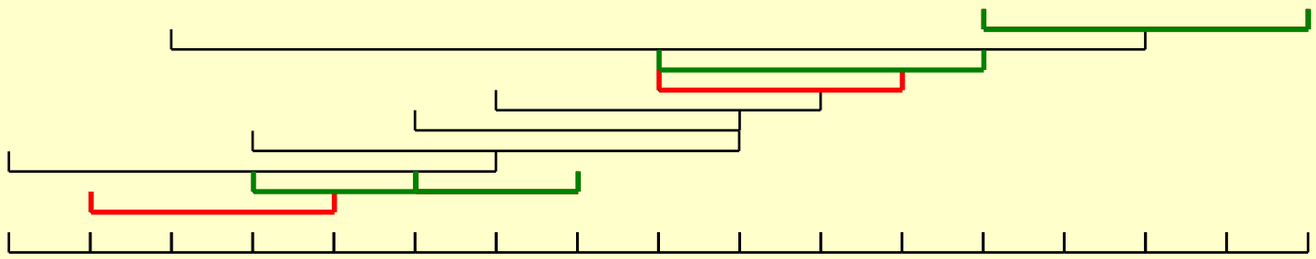
Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
	4	5	6	7	9	9	10	11	12	14	16

Discussion 12:
How can we be greedy?



☞ A DP Solution

☞ **A DP Solution** $a_1\ a_2\ \ldots a_i \ldots a_k\ \ldots a_j \ldots a_n$

☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \begin{cases} c_{ik} + c_{kj} + 1 & \texttt{if } S_{ij} \neq \Phi \end{cases}$$

☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \max_{a_k \in S_{ij}} \{ c_{ik} + c_{kj} + 1 \} \quad \texttt{if}\ S_{ij} \neq \Phi$$

☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \begin{cases} 0 & \texttt{if } S_{ij} = \Phi \\ \max\limits_{a_k \in S_{ij}} \{ c_{ik} + c_{kj} + 1 \} & \texttt{if } S_{ij} \neq \Phi \end{cases}$$

☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \begin{cases} 0 & \texttt{if } S_{ij} = \Phi \\ \max\limits_{a_k \in S_{ij}} \{ c_{ik} + c_{kj} + 1 \} & \texttt{if } S_{ij} \neq \Phi \end{cases}$$

$O(N^3)$

☞ **A DP Solution** $a_1\ a_2\ \ldots\ \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \begin{cases} 0 & \texttt{if } S_{ij} = \Phi \\ \max\limits_{a_k \in S_{ij}} \{ c_{ik} + c_{kj} + 1 \} & \texttt{if } S_{ij} \neq \Phi \end{cases}$$

O(N^3)

Can we be greedy?

☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \begin{cases} 0 & \texttt{if } S_{ij} = \Phi \\ \max\limits_{a_k \in S_{ij}} \{ c_{ik} + c_{kj} + 1 \} & \texttt{if } S_{ij} \neq \Phi \end{cases}$$

O(N^3)

Can we be greedy?

☞ **Greedy Rule 1: Select the interval which *starts earliest* (but not overlapping the already chosen intervals)**

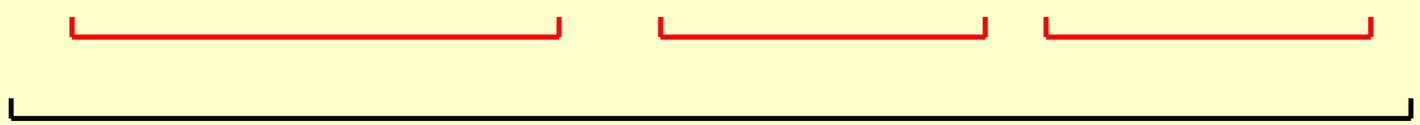
☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \begin{cases} 0 & \texttt{if } S_{ij} = \Phi \\ \max\limits_{a_k \in S_{ij}} \{ c_{ik} + c_{kj} + 1 \} & \texttt{if } S_{ij} \neq \Phi \end{cases}$$

O(N^3)

Can we be greedy?

☞ **Greedy Rule 1: Select the interval which *starts earliest* (but not overlapping the already chosen intervals)**



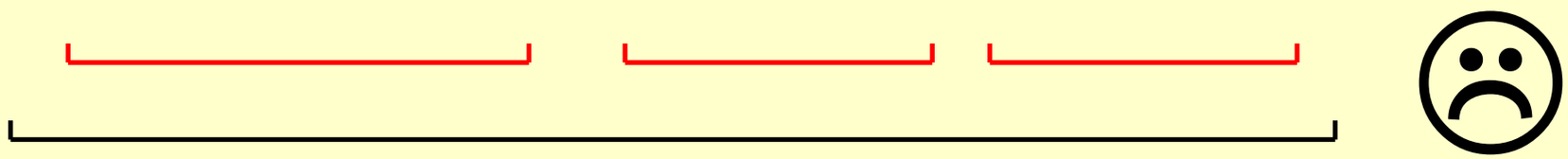
☞ **A DP Solution** $a_1\ a_2\ \ldots \underbrace{a_i \ldots a_k\ \ldots a_j}_{S_{ij}} \ldots a_n$

$$c_{ij} = \begin{cases} 0 & \texttt{if } S_{ij} = \Phi \\ \max\limits_{a_k \in S_{ij}} \{ c_{ik} + c_{kj} + 1 \} & \texttt{if } S_{ij} \neq \Phi \end{cases}$$

O(N^3)

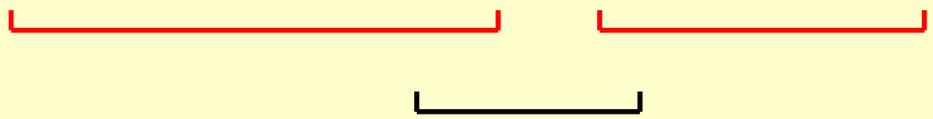
Can we be greedy?

☞ **Greedy Rule 1: Select the interval which *starts earliest* (but not overlapping the already chosen intervals)**

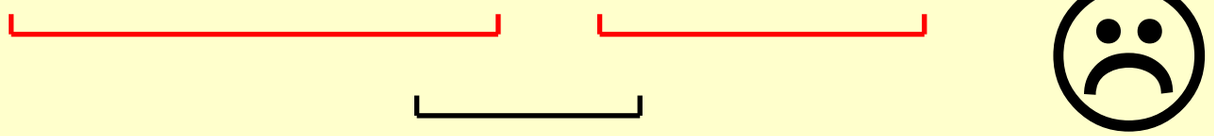


☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**

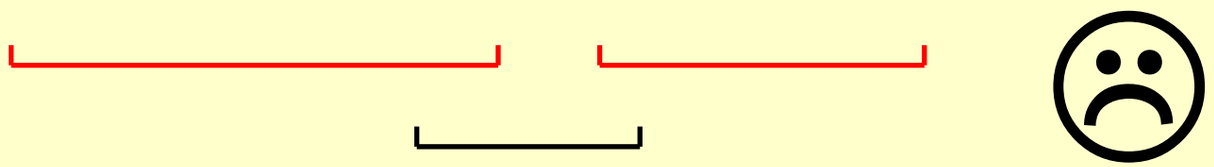
☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**

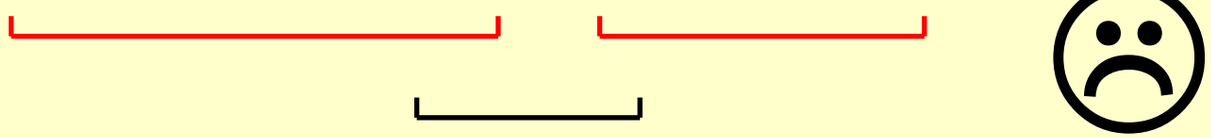


☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**

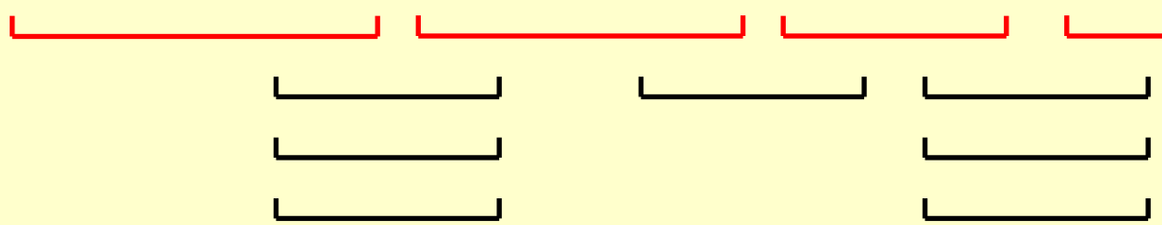


☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**

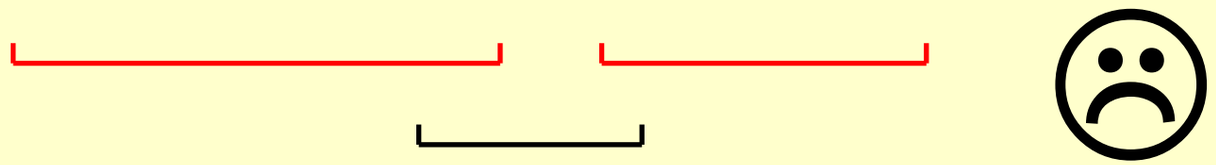
☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



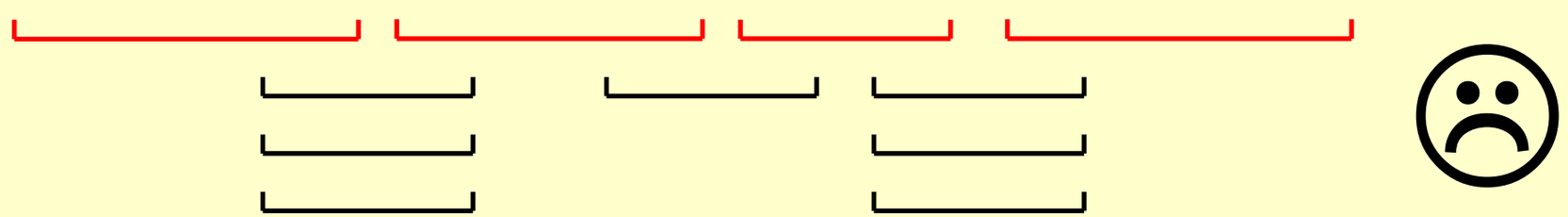
☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**



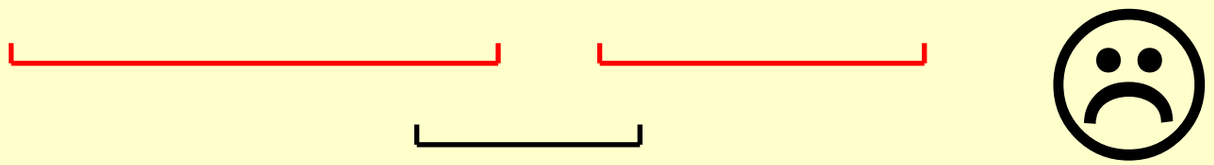
☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



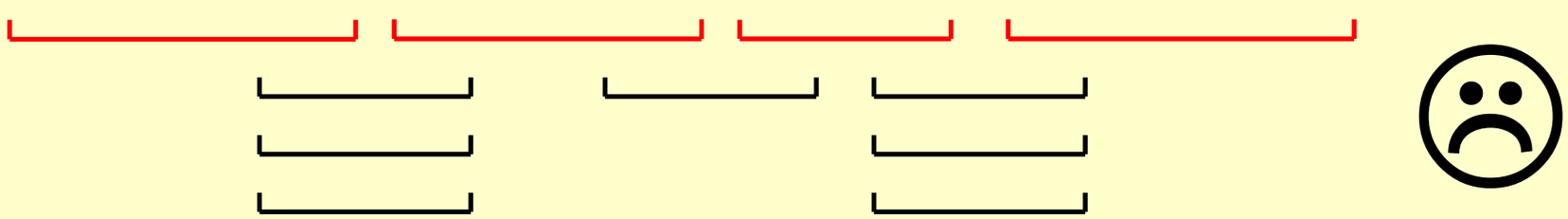
☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**



☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



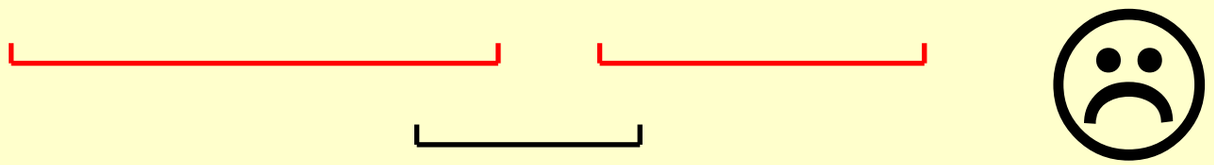
☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**



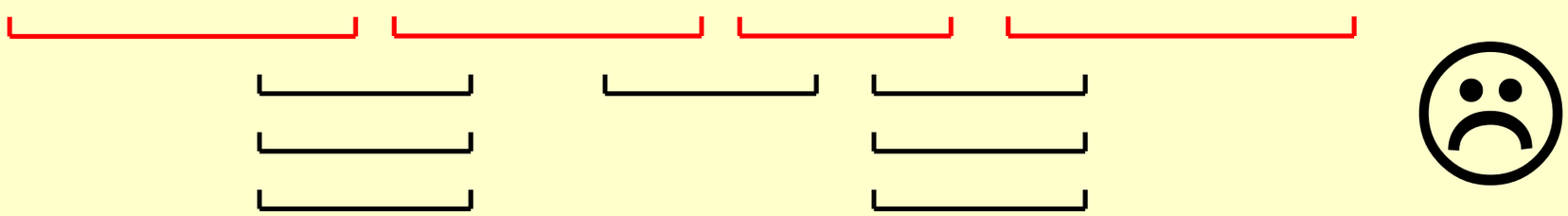
☞ **Greedy Rule 4: Select the interval which *ends first***

(but not overlapping the already chosen intervals)

☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**

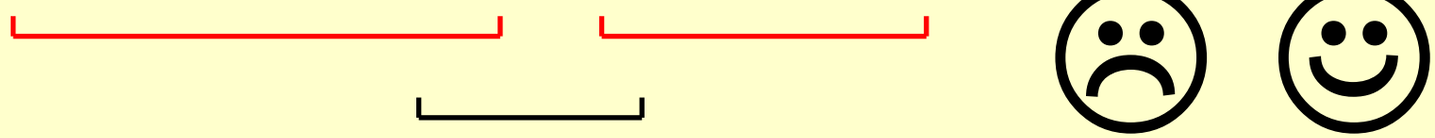


☞ **Greedy Rule 4: Select the interval which *ends first***

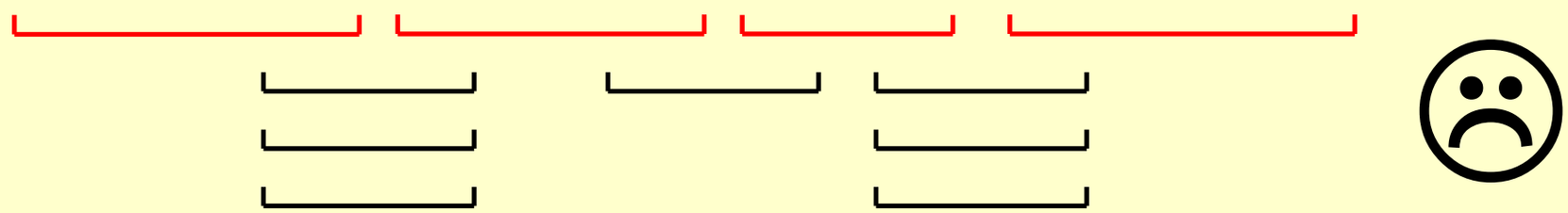
(but not overlapping the already chosen intervals)

Resource become free as soon as possible

☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**

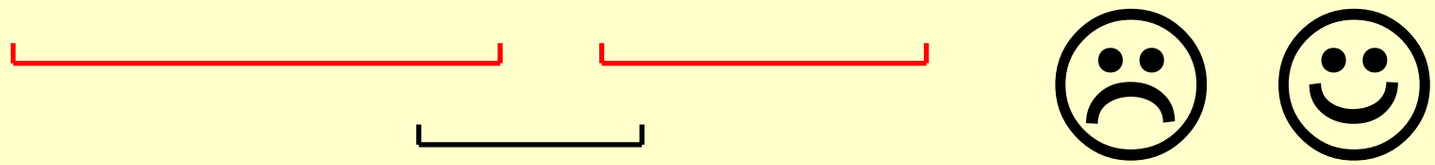


☞ **Greedy Rule 4: Select the interval which *ends first***

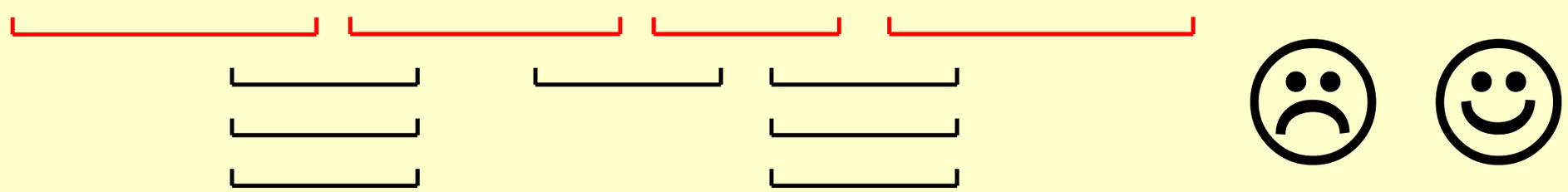
(but not overlapping the already chosen intervals)

Resource become free as soon as possible

☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**

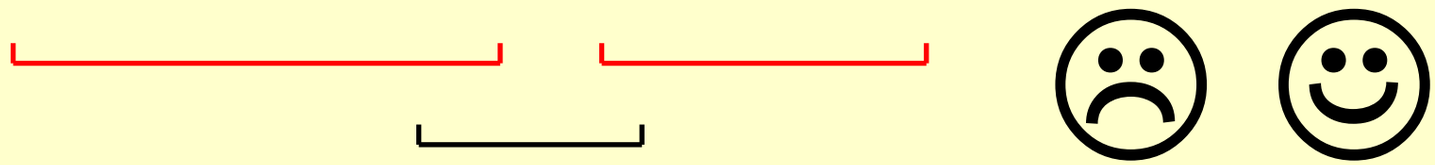


☞ **Greedy Rule 4: Select the interval which *ends first***

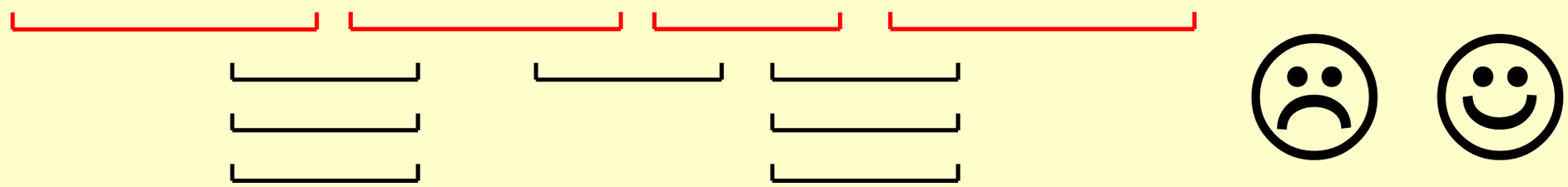
(but not overlapping the already chosen intervals)

Resource become free as soon as possible

☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



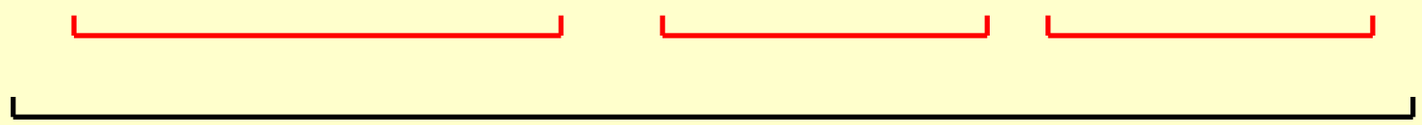
☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**



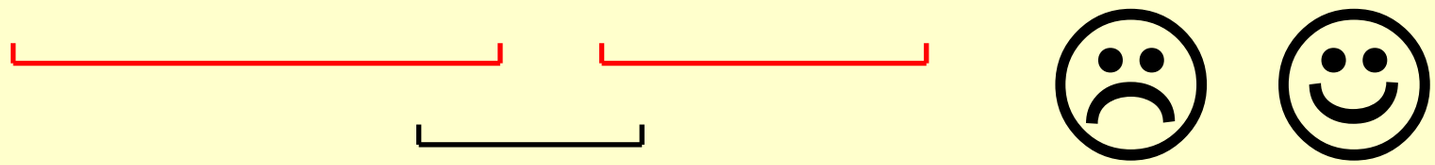
☞ **Greedy Rule 4: Select the interval which *ends first***

(but not overlapping the already chosen intervals)

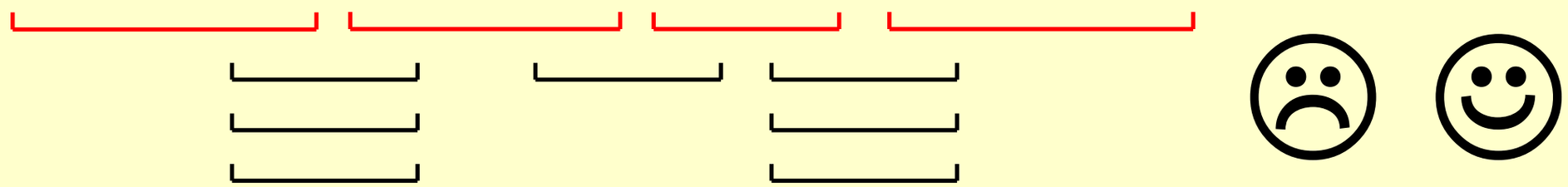
Resource become free as soon as possible



☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**



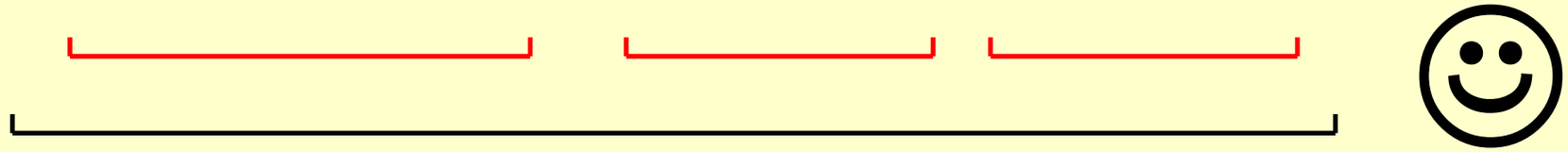
☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**



☞ **Greedy Rule 4: Select the interval which *ends first***

(but not overlapping the already chosen intervals)

Resource become free as soon as possible



Correctness:

① **Algorithm gives non-overlapping intervals**

② **The result is optimal**

Correctness:

① **Algorithm gives non-overlapping intervals**

② **The result is optimal**

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Correctness:

① **Algorithm gives non-overlapping intervals**

② **The result is optimal**

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof:

Correctness:

① **Algorithm gives non-overlapping intervals**

② **The result is optimal**

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

Correctness:

① Algorithm gives non-overlapping intervals

② The result is optimal

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

If a_m and a_{ef} are the same, we are done! Else ……

Correctness:

① Algorithm gives non-overlapping intervals

② The result is optimal

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

If a_m and a_{ef} are the same, we are done! Else ……

replace a_{ef} by a_m and get A_k'.

Correctness:

① Algorithm gives non-overlapping intervals

② The result is optimal

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

If a_m and a_{ef} are the same, we are done! Else ……

replace a_{ef} by a_m and get A_k'.

Since $f_m \le f_{ef}$, A_k' is another optimal solution.

Correctness:

① Algorithm gives non-overlapping intervals

② The result is optimal

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

If a_m and a_{ef} are the same, we are done! Else ……

replace a_{ef} by a_m and get A_k'.

Since $f_m \le f_{ef}$, A_k' is another optimal solution. ∎

Correctness:

① Algorithm gives non-overlapping intervals

② The result is optimal

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

If a_m and a_{ef} are the same, we are done! Else ……

replace a_{ef} by a_m and get A_k'.

Since $f_m \le f_{ef}$, A_k' is another optimal solution. ∎

Implementation:

① Select the first activity; Recursively solve for the rest.

Correctness:

① Algorithm gives non-overlapping intervals

② The result is optimal

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

If a_m and a_{ef} are the same, we are done! Else ……

replace a_{ef} by a_m and get A_k'.

Since $f_m \le f_{ef}$, A_k' is another optimal solution. ∎

Implementation:

① Select the first activity; Recursively solve for the rest.

② Remove tail recursion by iterations.

Correctness:

① Algorithm gives non-overlapping intervals

② The result is optimal

【Theorem】 Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof: Let A_k be the optimal solution set, and a_{ef} is the activity in A_k with the earliest finish time.

If a_m and a_{ef} are the same, we are done! Else ……

replace a_{ef} by a_m and get A_k'.

Since $f_m \le f_{ef}$, A_k' is another optimal solution. ∎

Implementation:

① Select the first activity; Recursively solve for the rest.

② Remove tail recursion by iterations. O(Nlog$N$)

☞ Another Look at DP Solution

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

If each activity has a weight …

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

If each activity has a weight …

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + w_j \} & \text{if } j > 1 \end{cases}$$

☞ Another Look at DP Solution

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + 1 \} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j.

If each activity has a weight …

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{ c_{1,j-1}, c_{1,k(j)} + w_j \} & \text{if } j > 1 \end{cases}$$

Q1: Is the DP solution still correct?
Q2: Is the Greedy solution still correct?

Elements of the Greedy Strategy

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution

Greedy Algorithms

- Active selection (aka interval scheduling)
- Huffman coding
- Minimum spanning trees
- Take-home messages

Huffman Codes – for file compression

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take ? bits to store the string as 1000 one-byte characters.

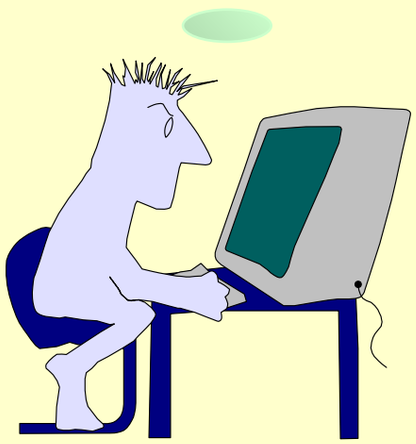
Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

Huffman Codes – for file compression

〖Example〗 **Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.**

Notice that we have only
4 distinct characters in that string.
Hence we need only
2 bits to identify them.



Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as $a = 00$, $u = 01$, $x = 10$, $z = 11$. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where C is the size of the character set */

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ **frequency ::=** number of occurrences of a symbol.

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ **frequency ::=** number of occurrences of a symbol.

In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ frequency ::= number of occurrences of a symbol.

In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

The size of the coded string can be reduced using variable-length codes, for example, *a* = 0, *u* = 110, *x* = 10, *z* = 111.

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ frequency ::= number of occurrences of a symbol.

In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

The size of the coded string can be reduced using variable-length codes, for example, *a* = 0, *u* = 110, *x* = 10, *z* = 111. ➡ 0001011001011 1

Huffman Codes – for file compression

〖Example〗 Suppose our text is a string of length 1000 that comprises the characters *a*, *u*, *x*, and *z*. Then it will take 8000 bits to store the string as 1000 one-byte characters.

We may encode the symbols as *a* = 00, *u* = 01, *x* = 10, *z* = 11. For example, *aaaxuaxz* is encoded as 0000001001001011. Then the space taken by the string with length 1000 will be 2000 bits + space for code table. /* $\lceil \log C \rceil$ bits are needed in a standard encoding where *C* is the size of the character set */

➢ frequency ::= number of occurrences of a symbol.

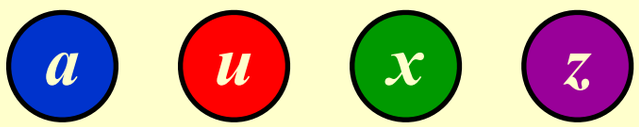
In string *aaaxuaxz* , $f(a) = 4$, $f(u) = 1$, $f(x) = 2$, $f(z) = 1$.

The size of the coded string can be reduced using variable-length codes, for example, *a* = 0, *u* = 110, *x* = 10, *z* = 111. ➡ 0001011001011 1

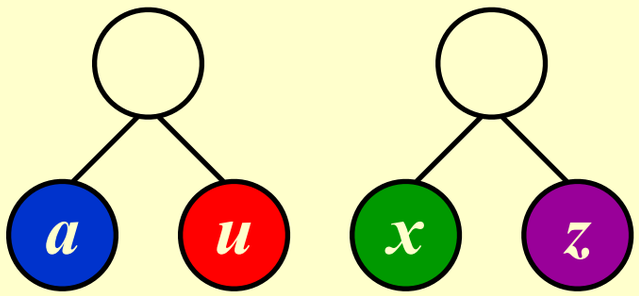
Note: If all the characters occur with the same frequency, then there are not likely to be any savings.

Representation of the original code in a binary tree /* trie */

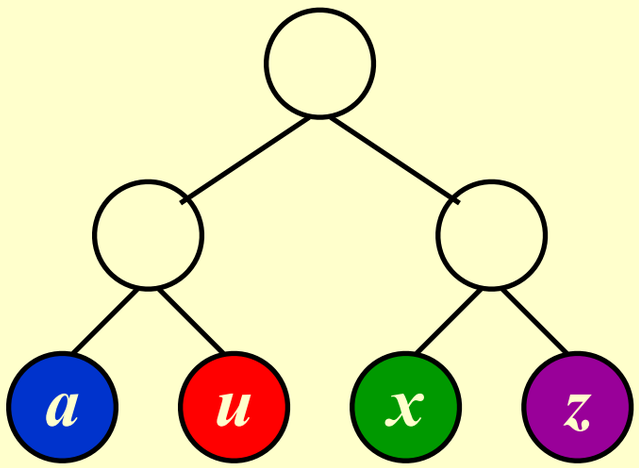
Representation of the original code in a binary tree /* trie */



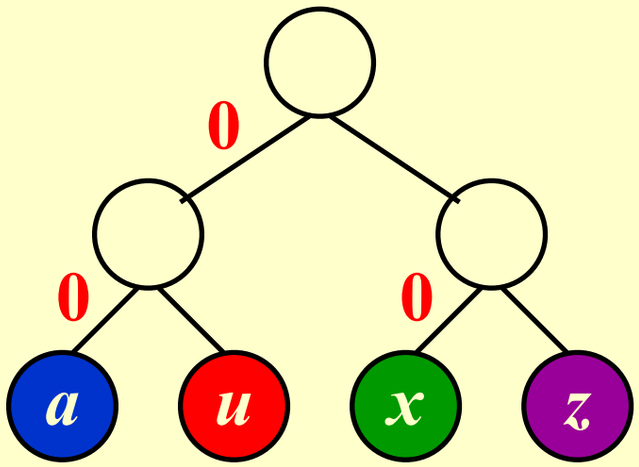
Representation of the original code in a binary tree /* trie */



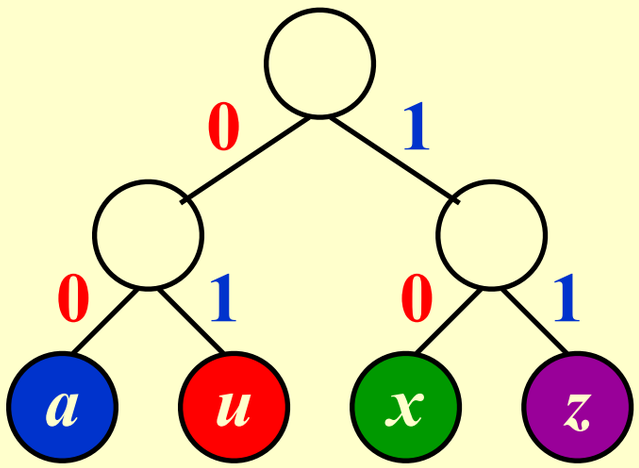
Representation of the original code in a binary tree /* trie */



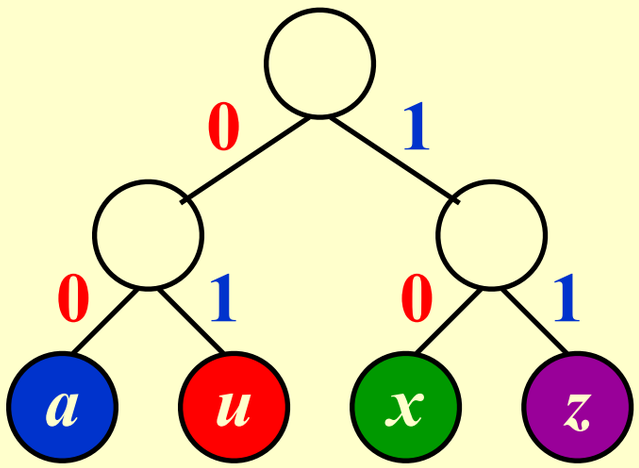
Representation of the original code in a binary tree /* trie */



Representation of the original code in a binary tree /* trie */

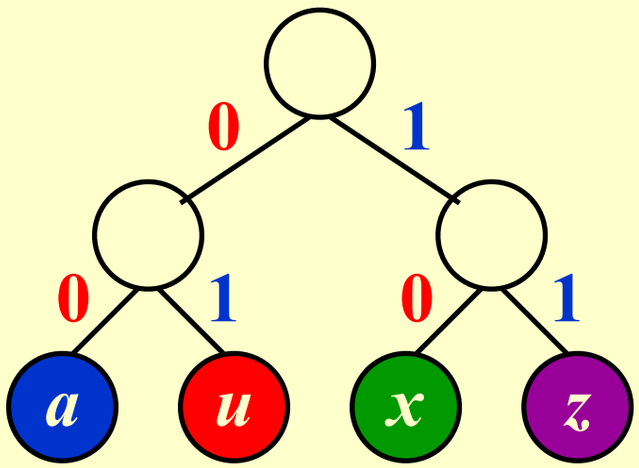


Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

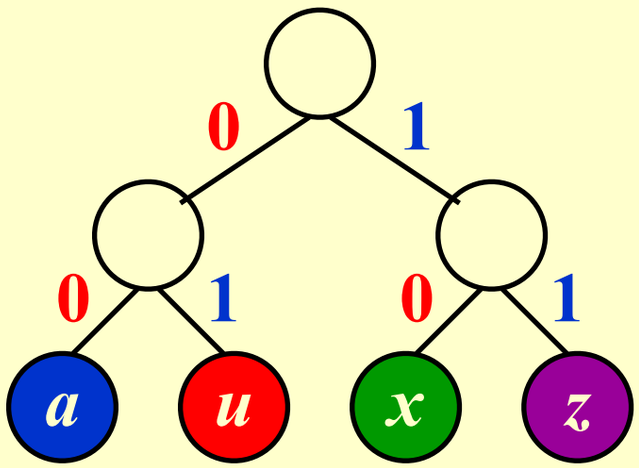
Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

Cost **(*aaaxuaxz* → 00000010010011011)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the original code in a binary tree /* trie */

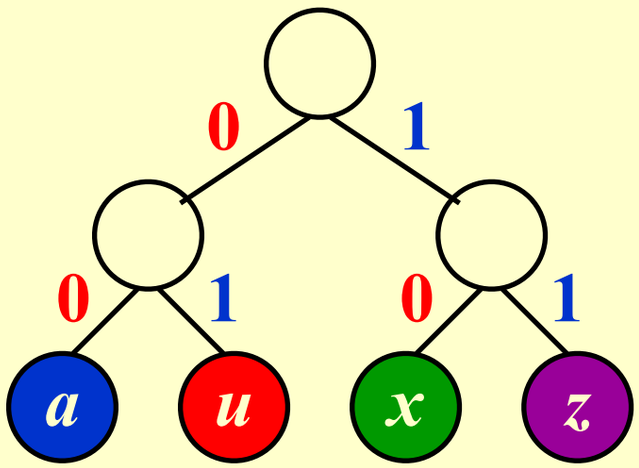


➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

Cost **(*aaaxuaxz* → 00000010010011)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

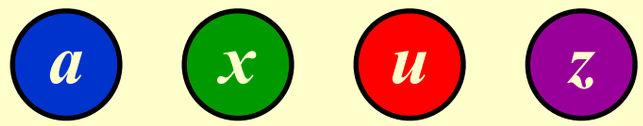
Representation of the original code in a binary tree /* trie */



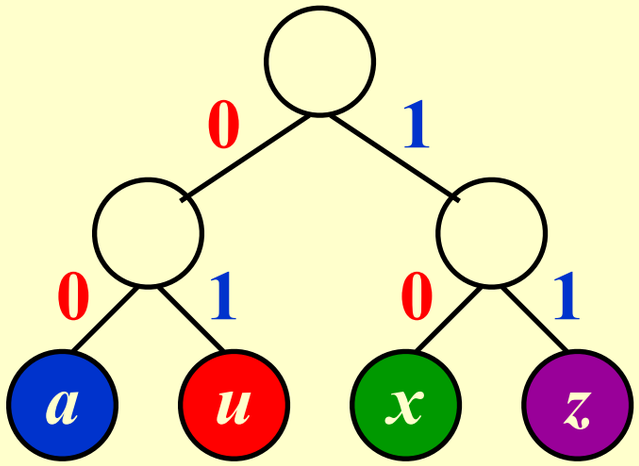
➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

Cost **(*aaaxuaxz* → 00000010010011)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



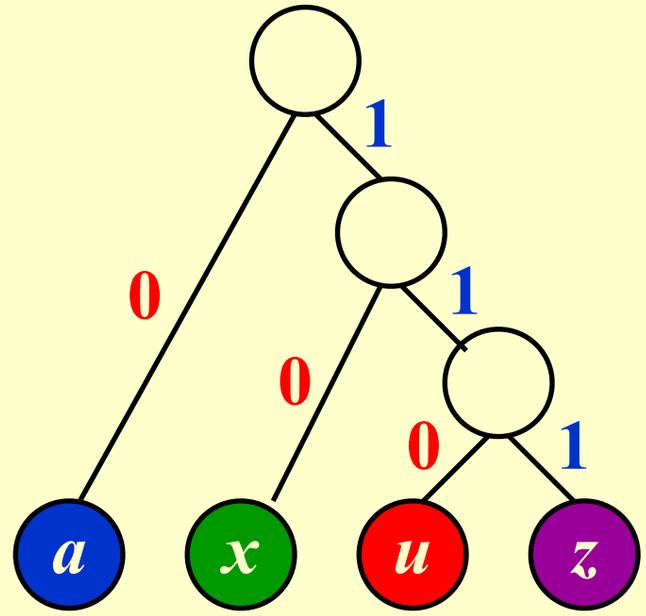
Representation of the original code in a binary tree /* trie */



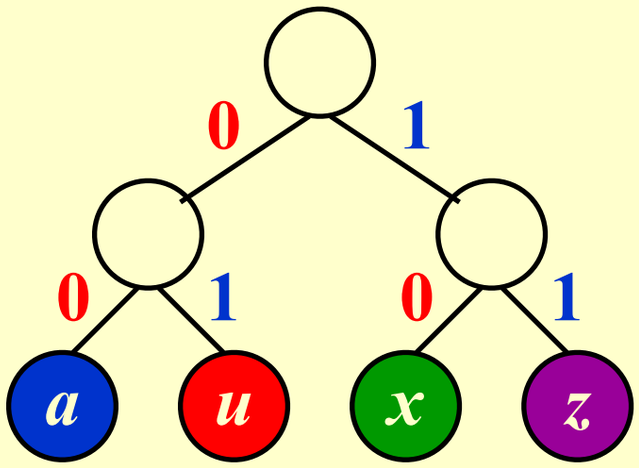
➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



Representation of the original code in a binary tree /* trie */

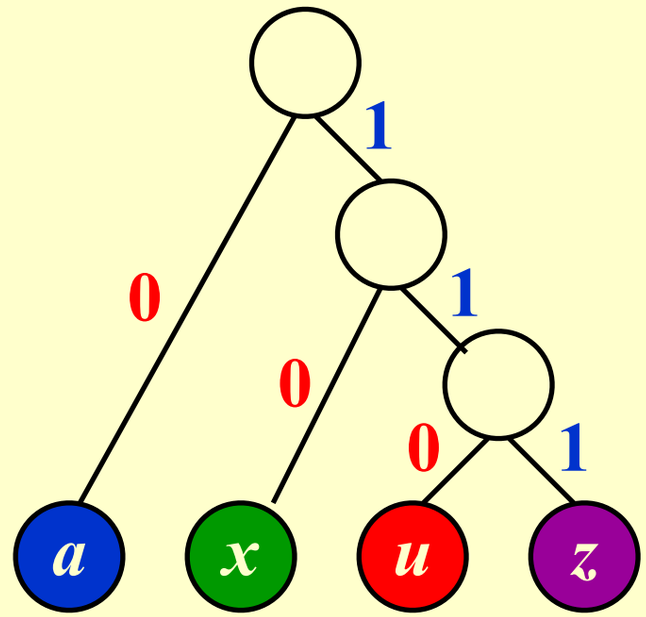


➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

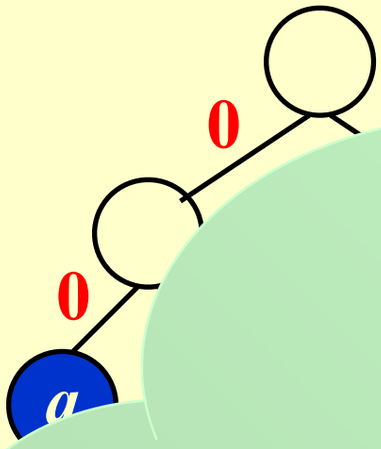
Cost **(*aaaxuaxz* → 000000100100101 1)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

Cost **(*aaaxuaxz* → 000101100010111)**
= 1×4 + 3×1 + 2×2 + 3×1 = 14



Representation of the original code in a binary tree /* trie */



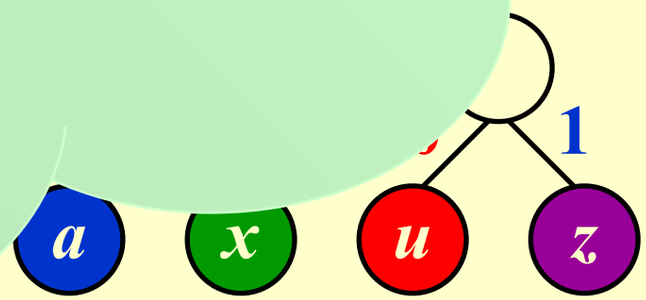
➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code is $\sum_i d_i f_i$.**

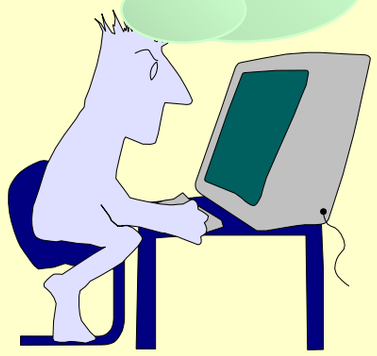
001011)

Now, with

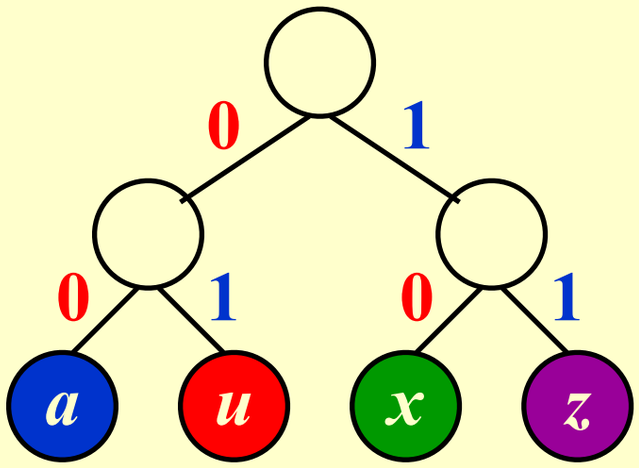
$a = 0$, $u = 110$, $x = 10$, $z = 111$

and the string 000101100101111, can you decode it?





Representation of the original code in a binary tree /* trie */

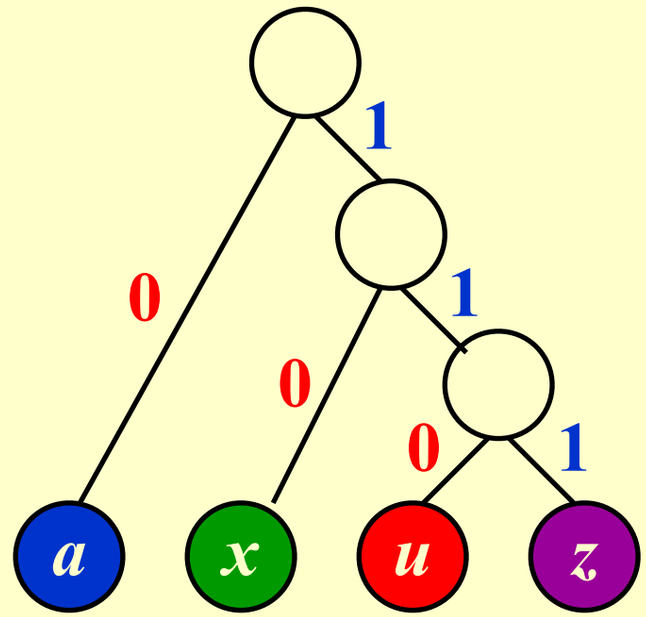


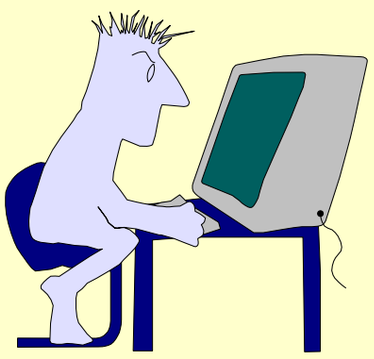
➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

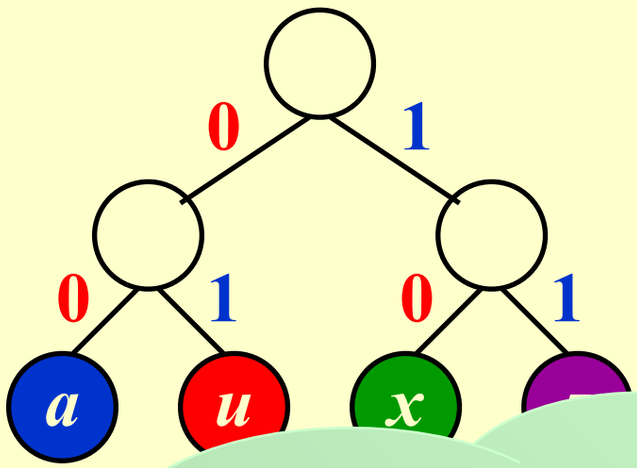
Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

Representation of the optimal code in a binary tree





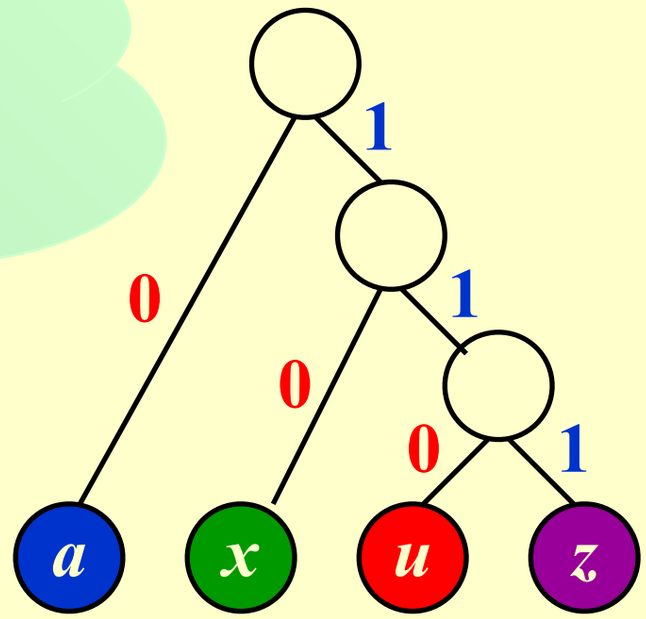
Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

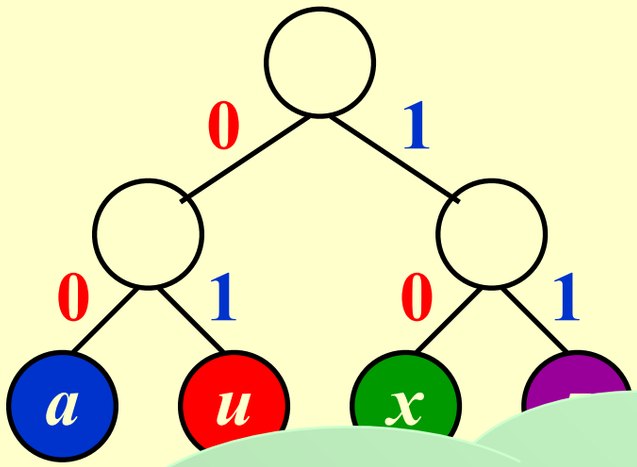
Cost **(*aaaxuaxz* → 00000010010011 1)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



The answer is *aaaxuaxz* (with *a* = 0, *u* = 110, *x* = 10, *z* = 111). What makes this decoding method work?

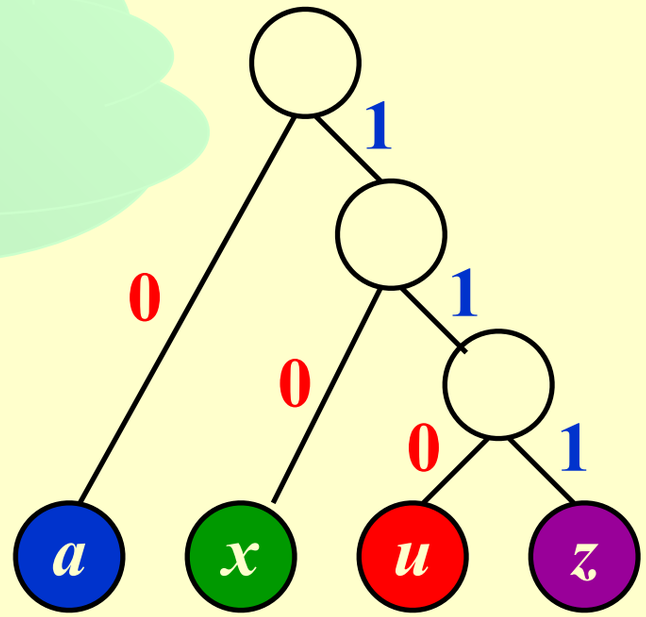
Representation of the original code in a binary tree /* trie */



➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

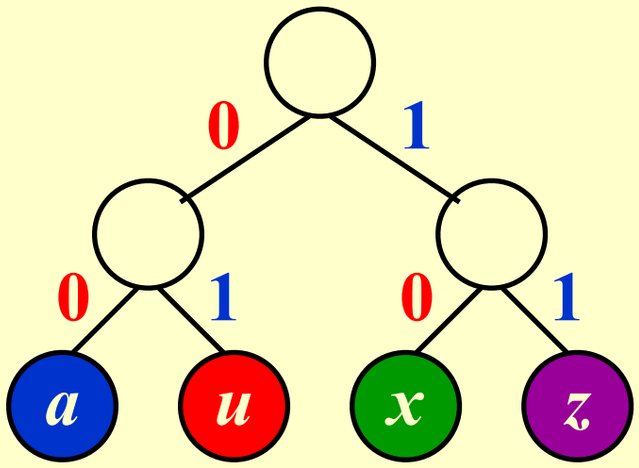
Cost (*aaaxuaxz* → 00000010010011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree



The trick is:
No code is a *prefix* of another.

Representation of the original code in a binary tree /* trie */

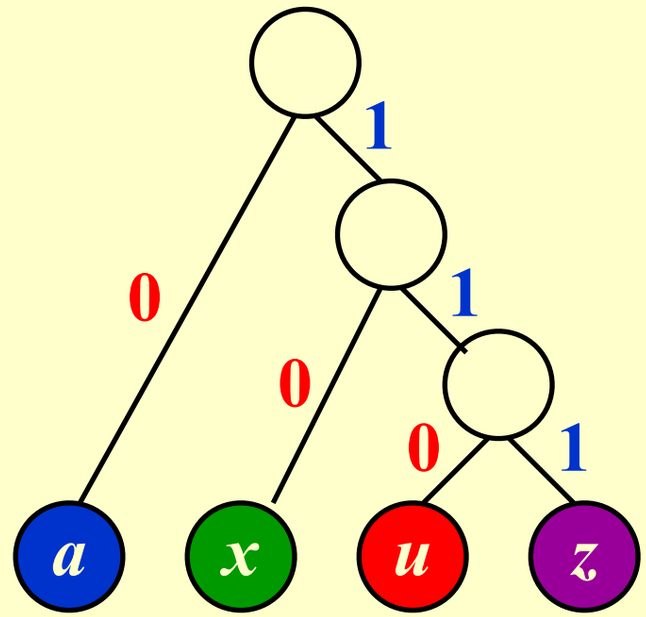


➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

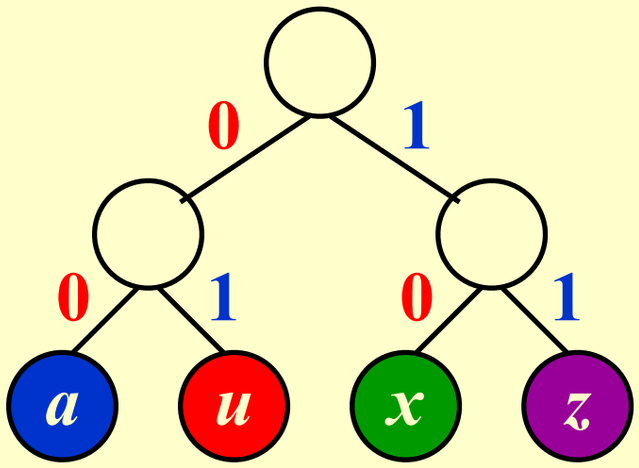
Cost **(*aaaxuaxz* → 000000100100101 1)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

Cost **(*aaaxuaxz* → 000101100010111)**
= 1×4 + 3×1 + 2×2 + 3×1 = 14



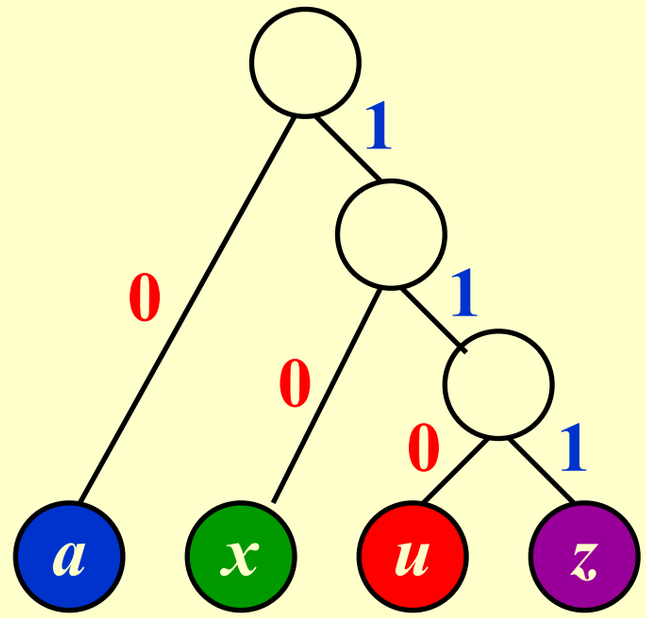
Representation of the original code in a binary tree /* trie */



➢ **If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.**

Cost **(*aaaxuaxz* → 000000100100101 1)**
= 2×4 + 2×1 + 2×2 + 2×1 = 16

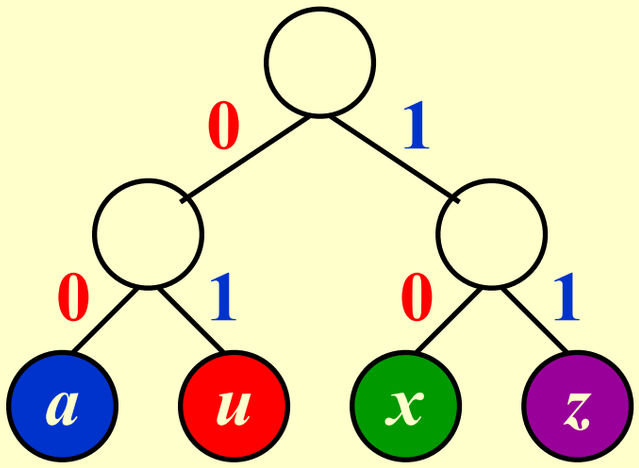
Representation of the optimal code in a binary tree



Cost **(*aaaxuaxz* → 000101100101 11)**
= 1×4 + 3×1 + 2×2 + 3×1 = 14

Discussion 13: **What must the tree look like if we are to decode unambiguously?**

Representation of the original code in a binary tree /* trie */

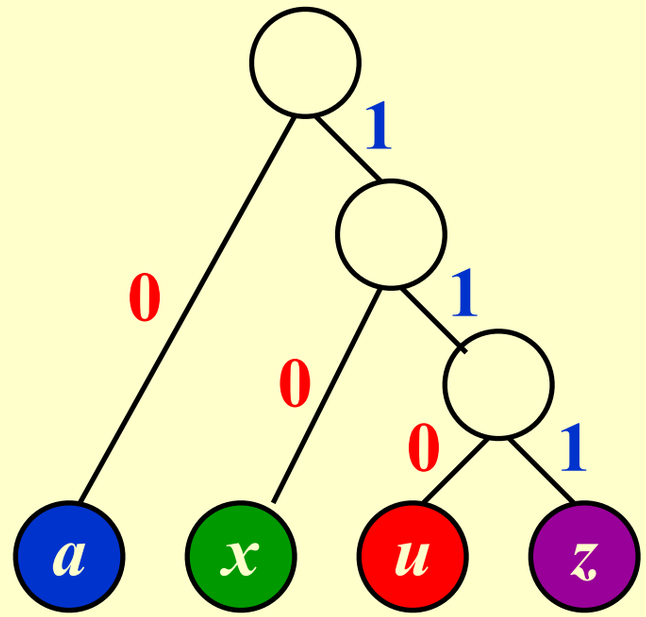


➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

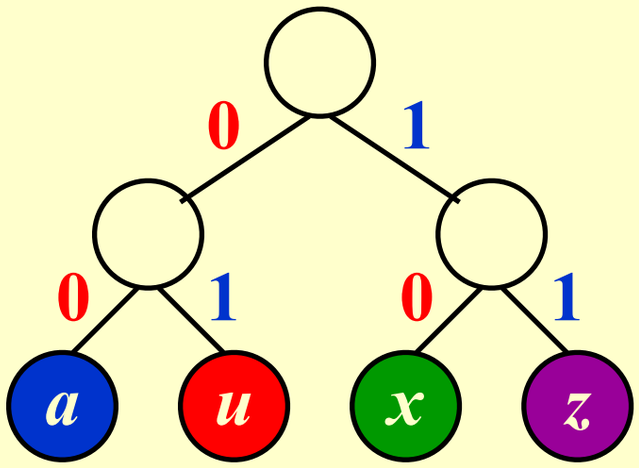
Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

Representation of the optimal code in a binary tree



Any sequence of bits can always be decoded unambiguously if the characters are placed only at the leaves of a *full tree* – such kind of code is called *prefix code*.

Representation of the original code in a binary tree /* trie */

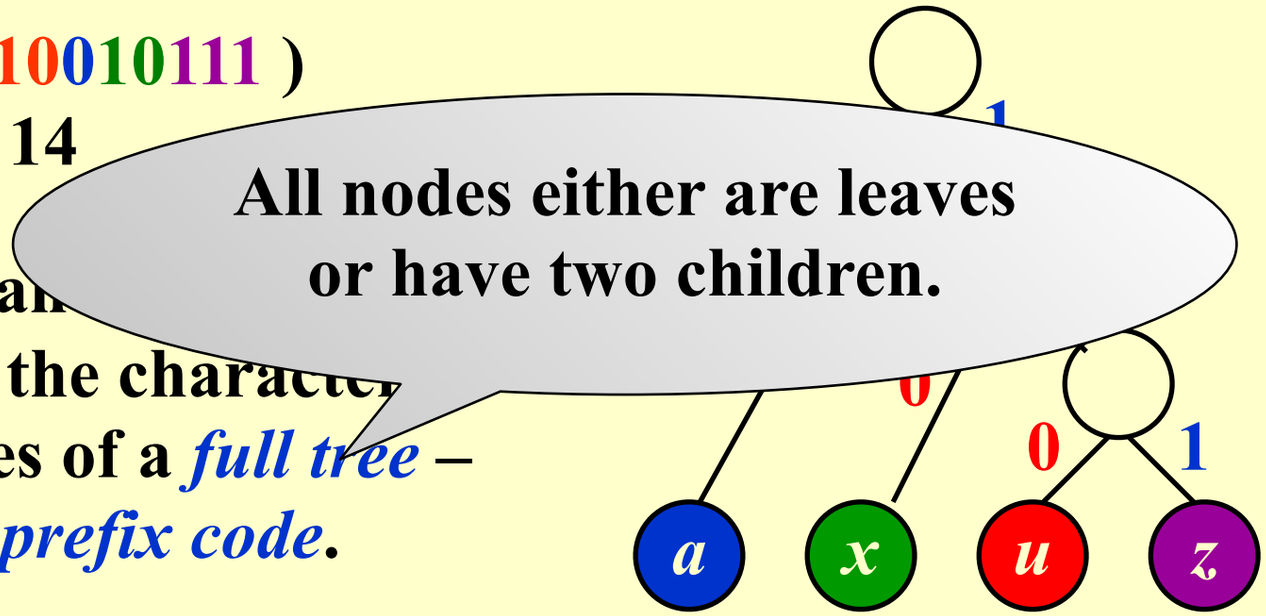


➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

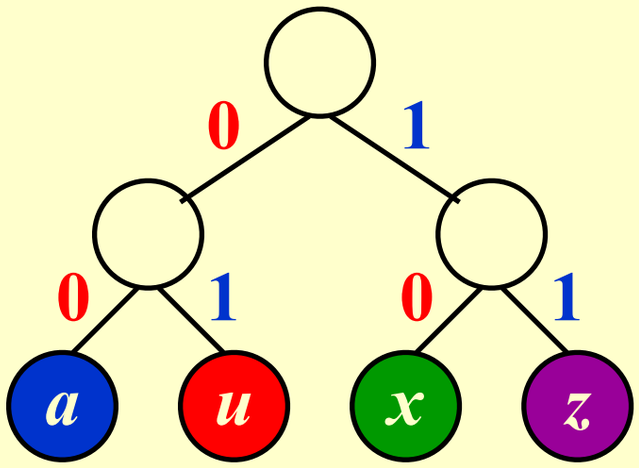
Representation of the optimal code in a binary tree

Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14



Any sequence of bits can be decoded unambiguously if the characters are placed only at the leaves of a *full tree* – such kind of code is called *prefix code*.

Representation of the original code in a binary tree /* trie */



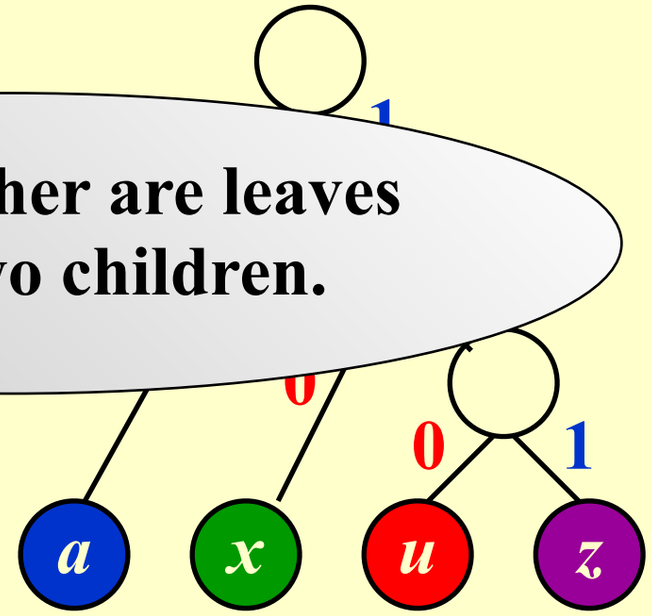
➢ If character C_i is at depth d_i and occurs f_i times, then the cost of the code = $\Sigma\, d_i f_i$.

Cost (*aaaxuaxz* → 00000010010011011)
= 2×4 + 2×1 + 2×2 + 2×1 = 16

Representation of the optimal code in a binary tree

Cost (*aaaxuaxz* → 00010110010111)
= 1×4 + 3×1 + 2×2 + 3×1 = 14

All nodes either are leaves or have two children.



Any sequence of bits can be decoded unambiguously if the characters are placed only at the leaves of a *full tree* – such kind of code is called *prefix code*.

Find the full binary tree of minimum total cost where all characters are contained in the leaves.

➢ Huffman's Algorithm (1952)

➢ Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{   consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
     }
}
```

➢ Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{   consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
     }
}
```

$$T = \mathrm{O}(\quad ? \quad)$$

➢ Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{   consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
     }
}
```

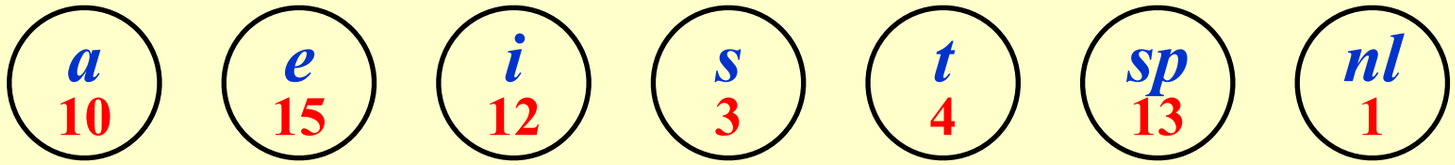
$$T = \mathrm{O}(\quad C \log C \quad)$$

〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1

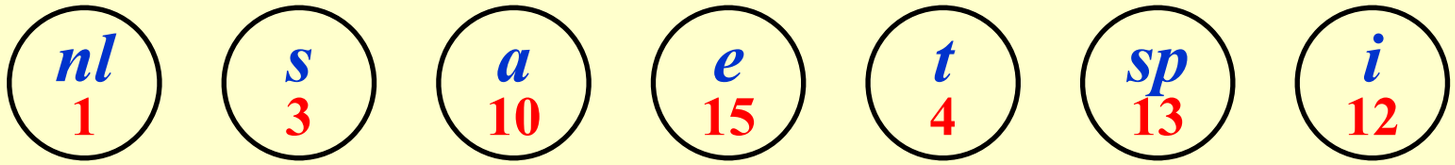
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



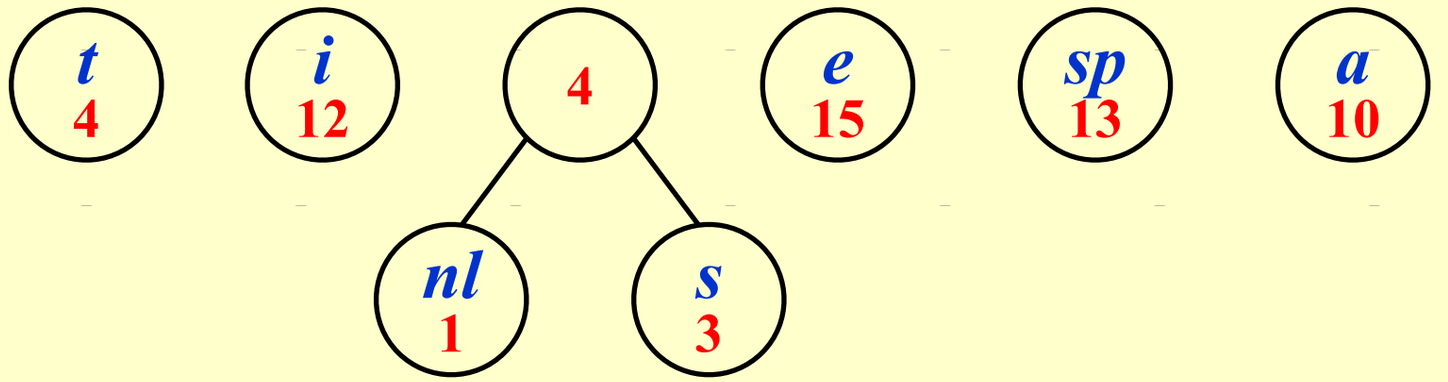
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



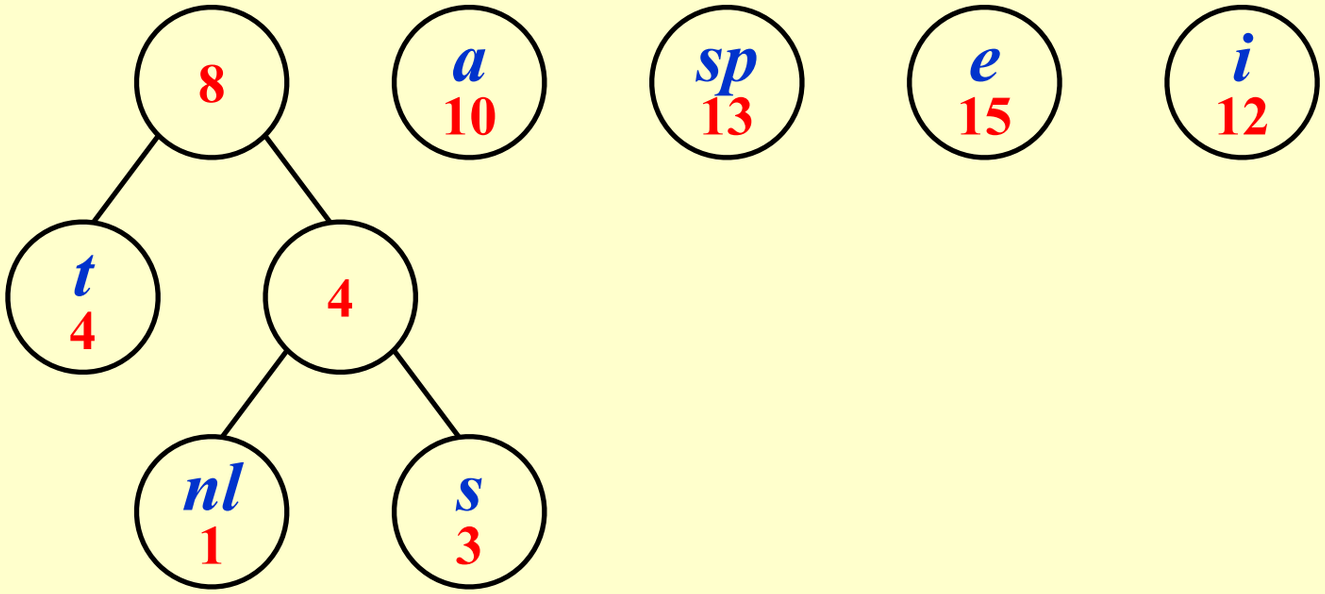
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



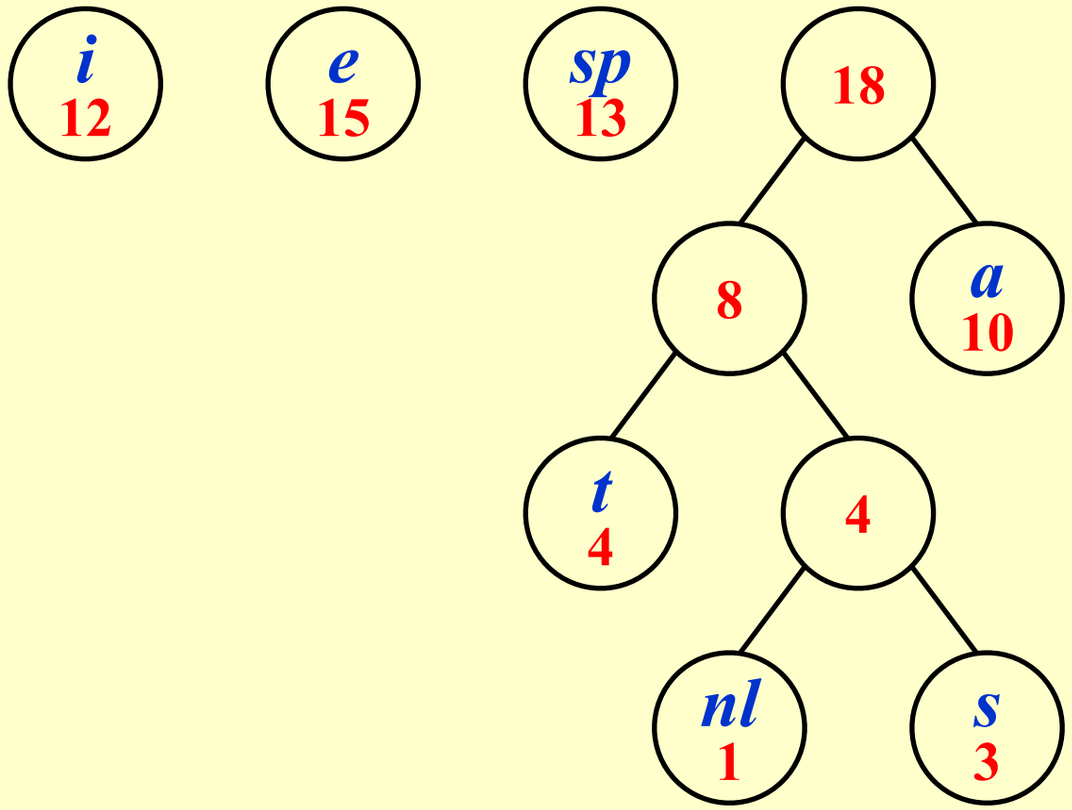
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



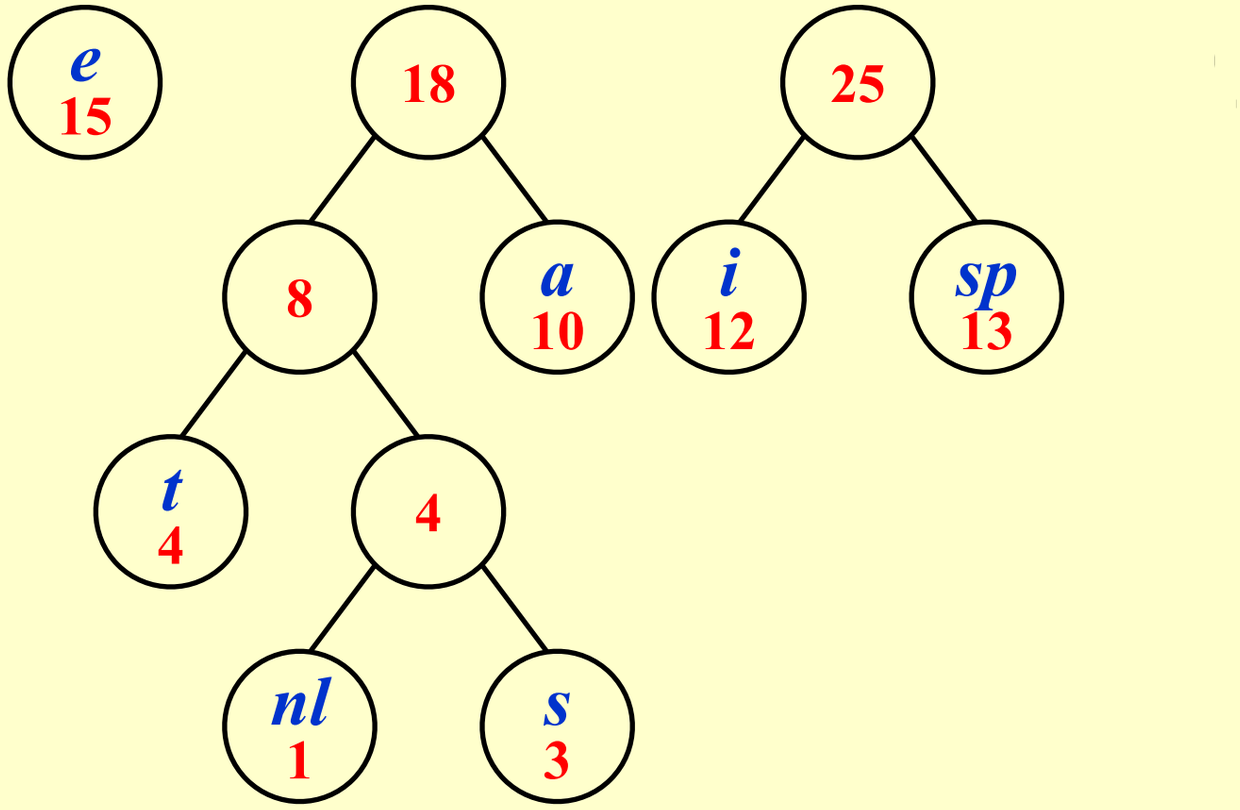
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



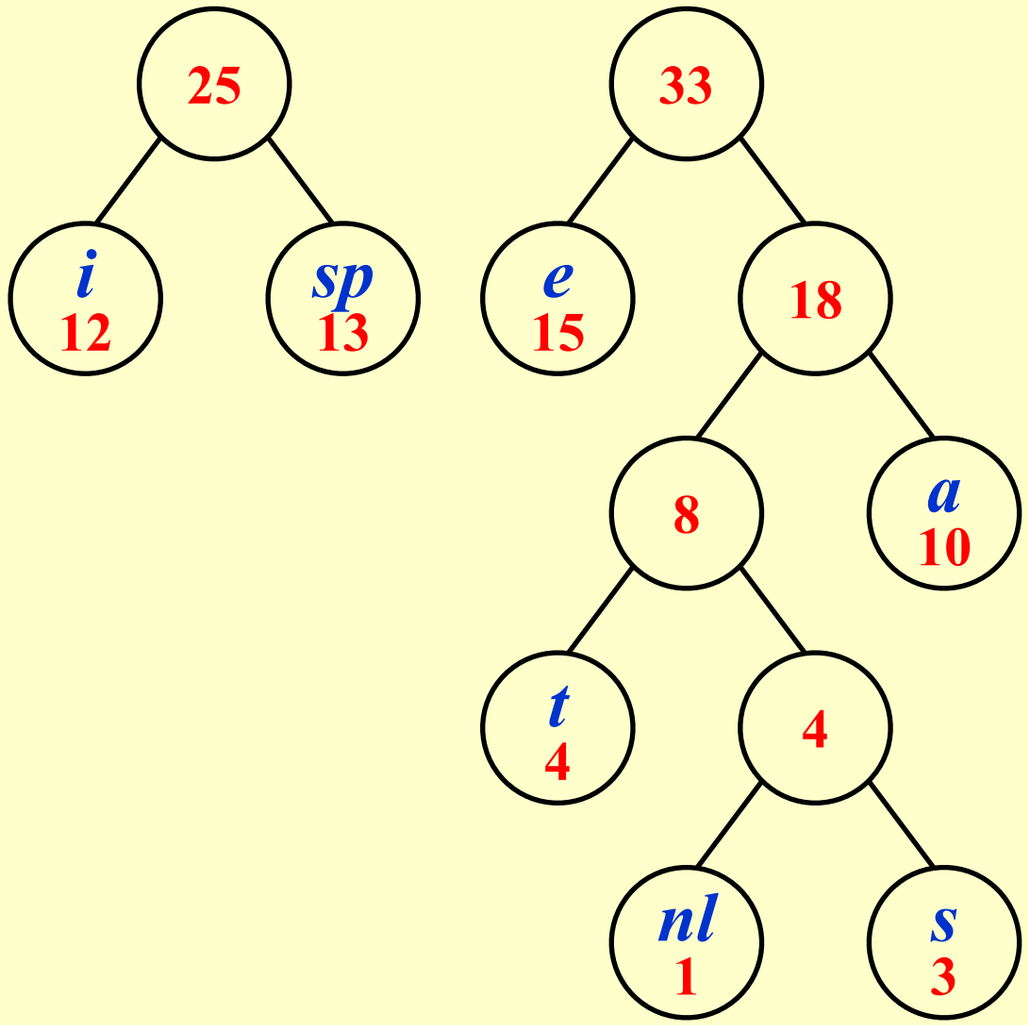
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



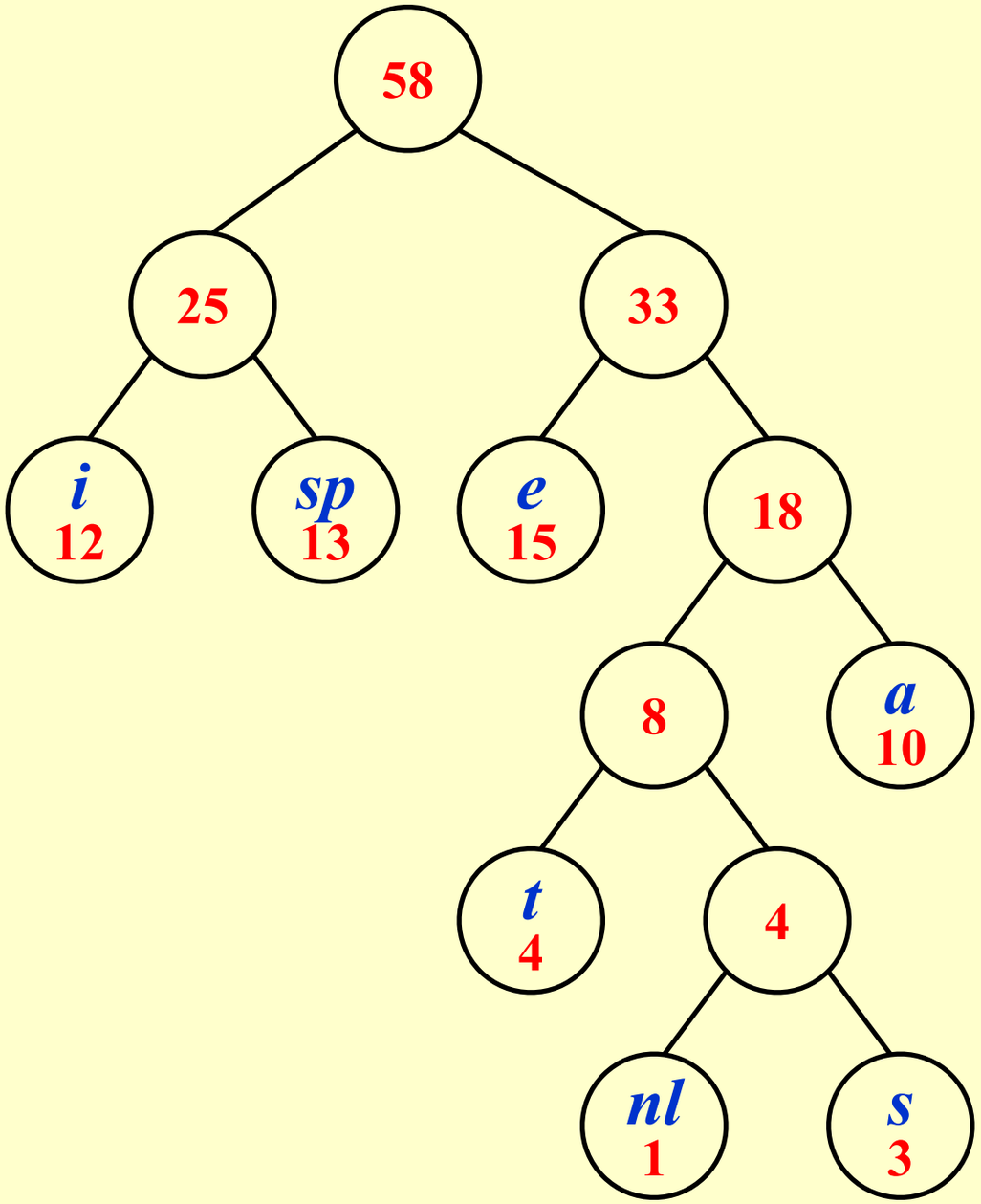
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



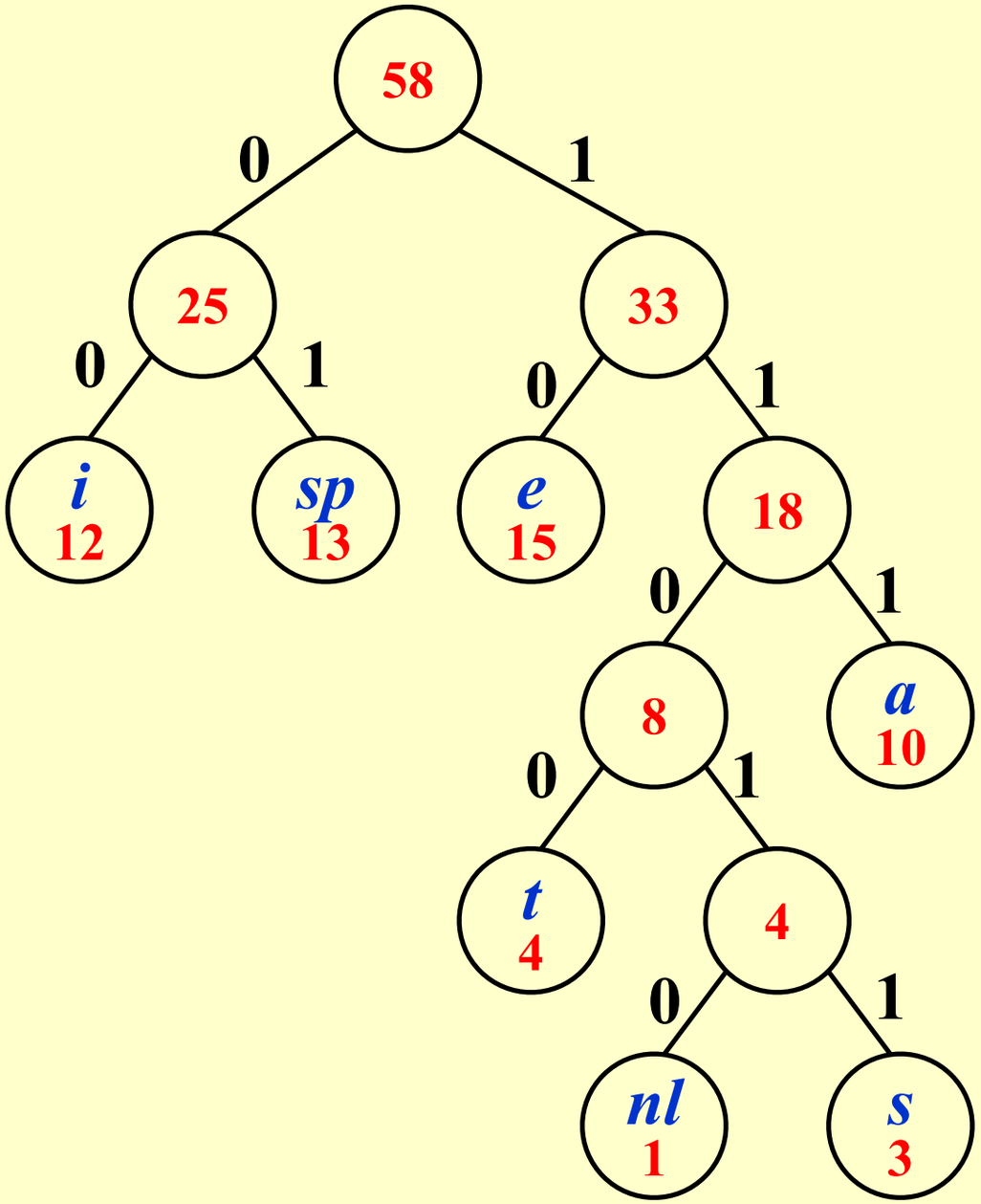
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



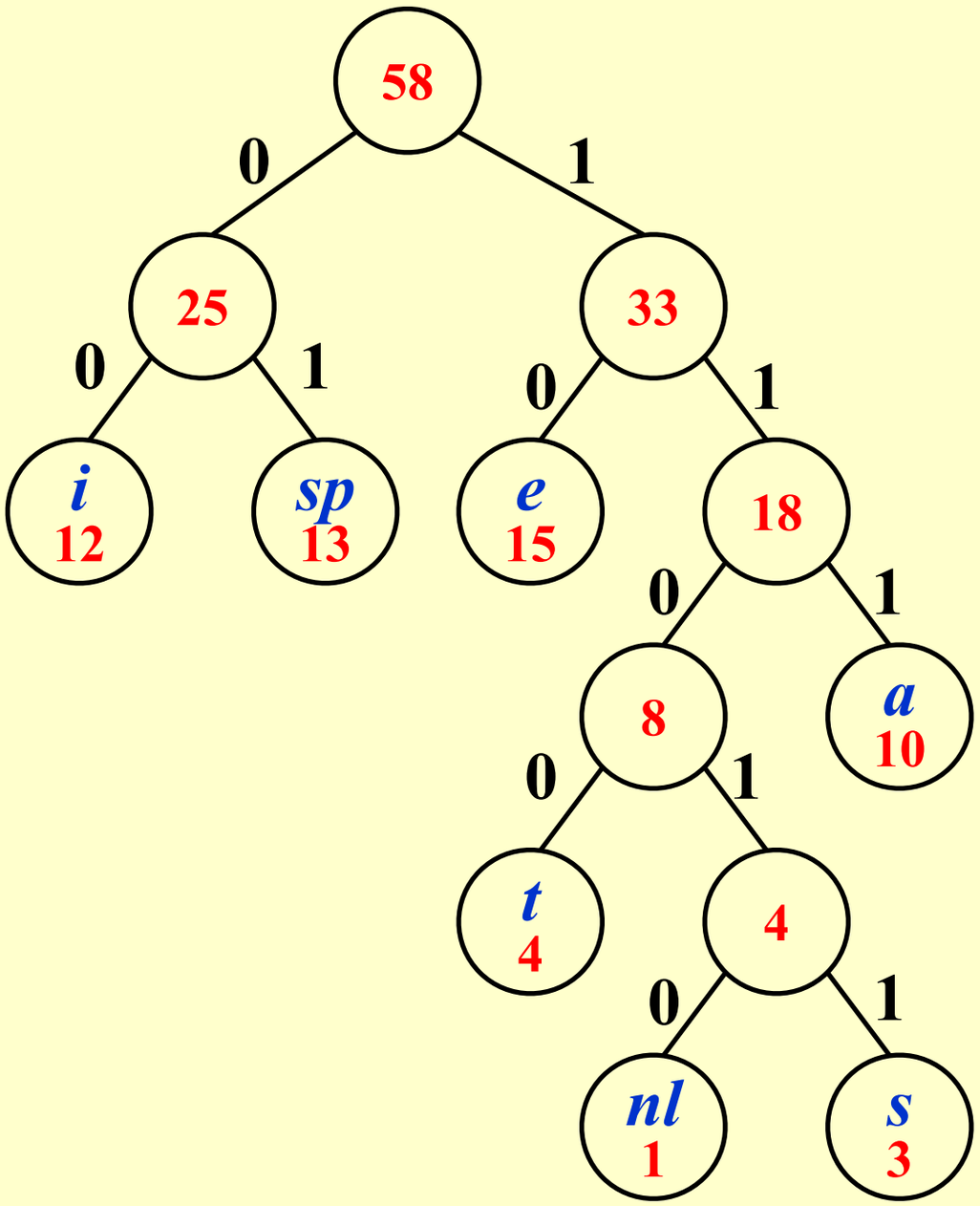
〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



〖Example〗

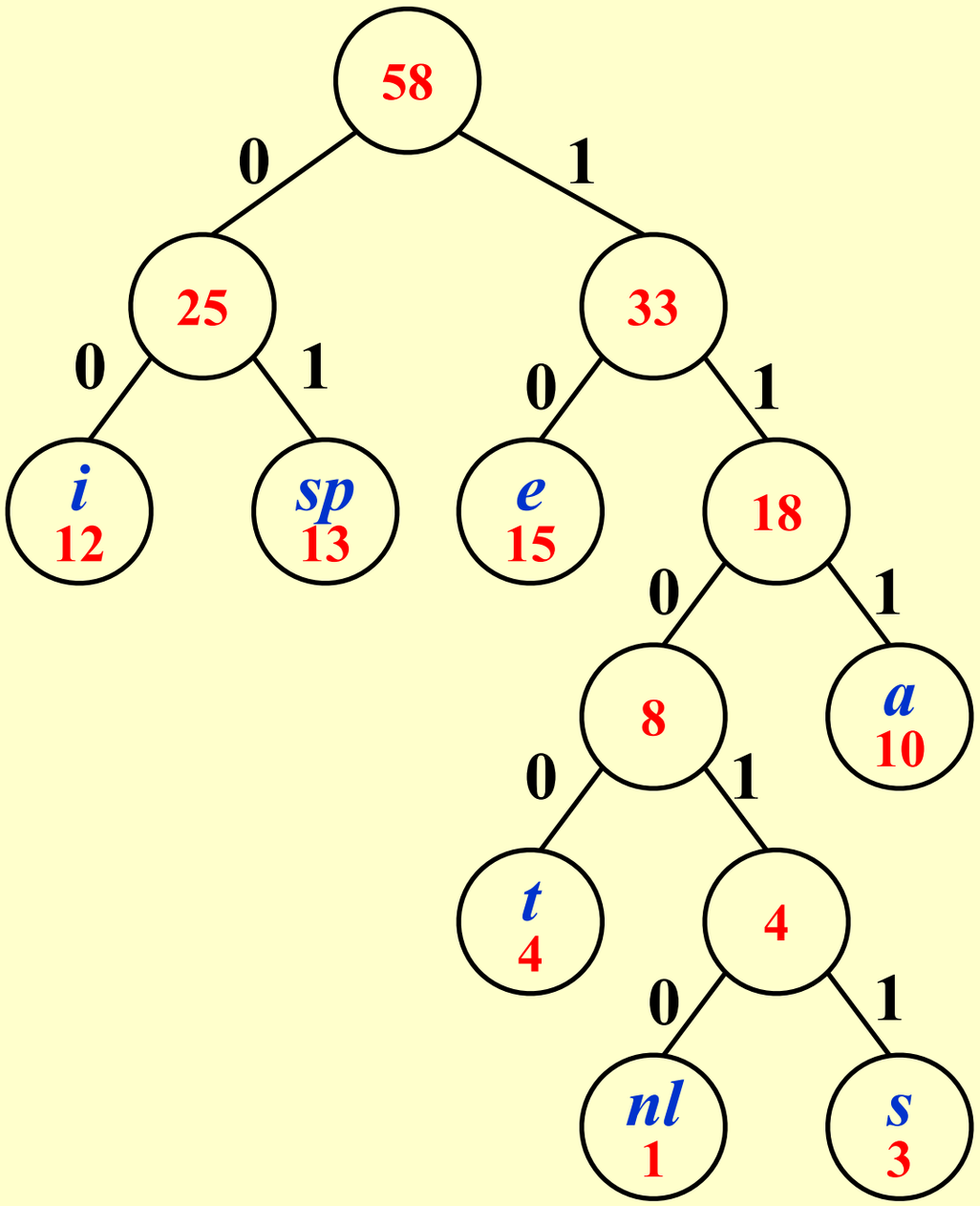
C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



a : 111
e : 10
i : 00
s : 11011
t : 1100
sp : 01
nl : 11010

〖Example〗

C_i	*a*	*e*	*i*	*s*	*t*	*sp*	*nl*
f_i	10	15	12	3	4	13	1



a : 111
e : 10
i : 00
s : 11011
t : 1100
sp : 01
nl : 11010

$$\begin{aligned} Cost &= 3\times10 + 2\times15 \\ &\quad + 2\times12 + 5\times3 \\ &\quad + 4\times4 + 2\times13 \\ &\quad + 5\times1 \\ &= 146 \end{aligned}$$

Correctness:

① The greedy-choice property

Correctness:

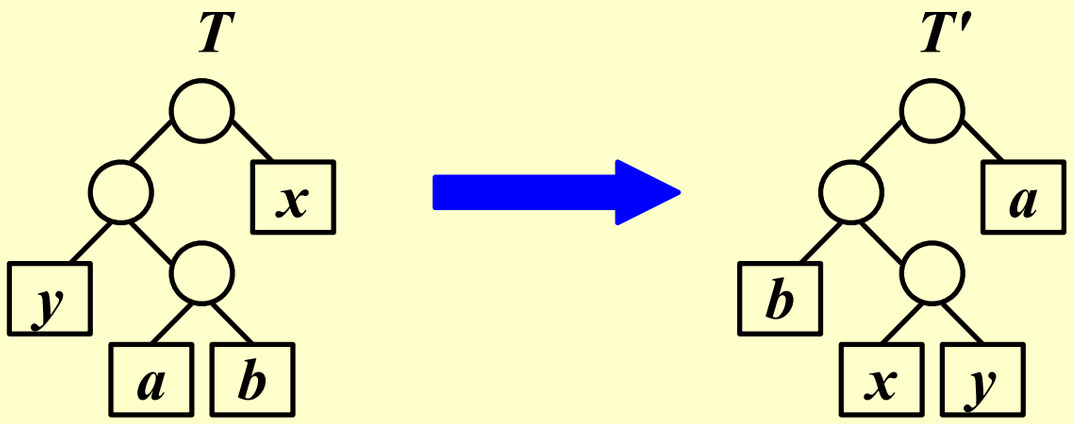
① The greedy-choice property

【Lemma】 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Correctness:

① The greedy-choice property

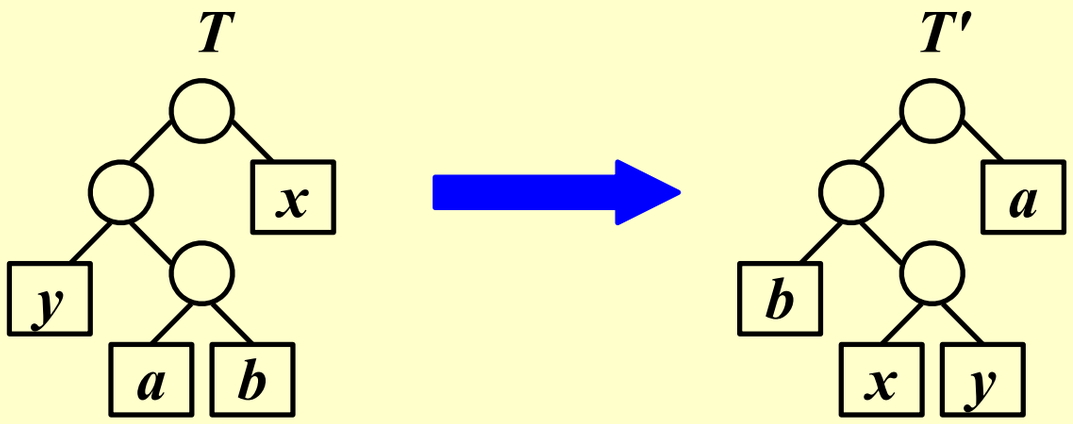
【Lemma】 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



Correctness:

① The greedy-choice property

【Lemma】 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



$$\text{Cost}(T') \le \text{Cost}(T)$$

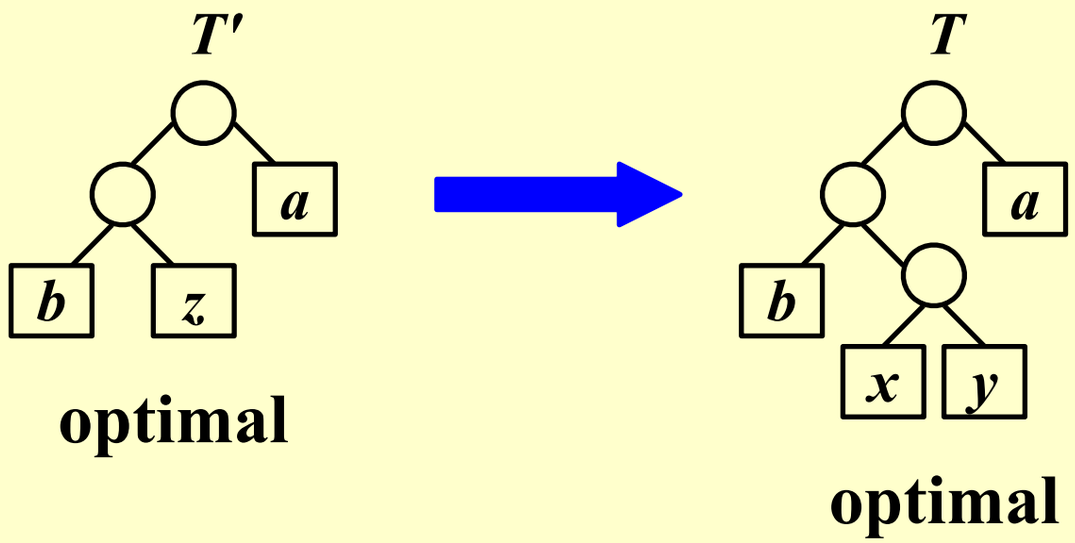
② The optimal substructure property

② The optimal substructure property

【Lemma】 Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

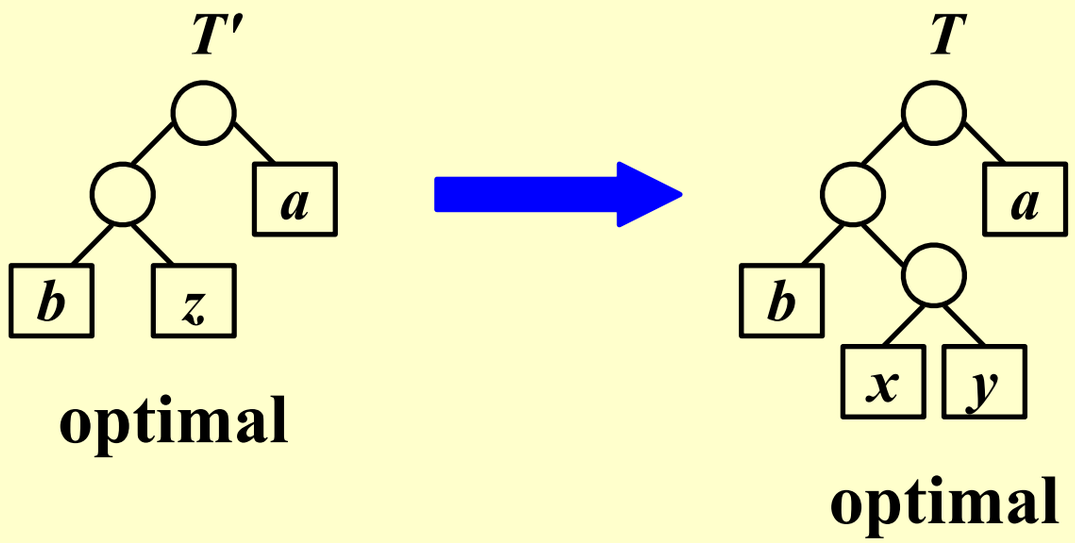
② The optimal substructure property

【Lemma】 Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.



② The optimal substructure property

【Lemma】 Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.



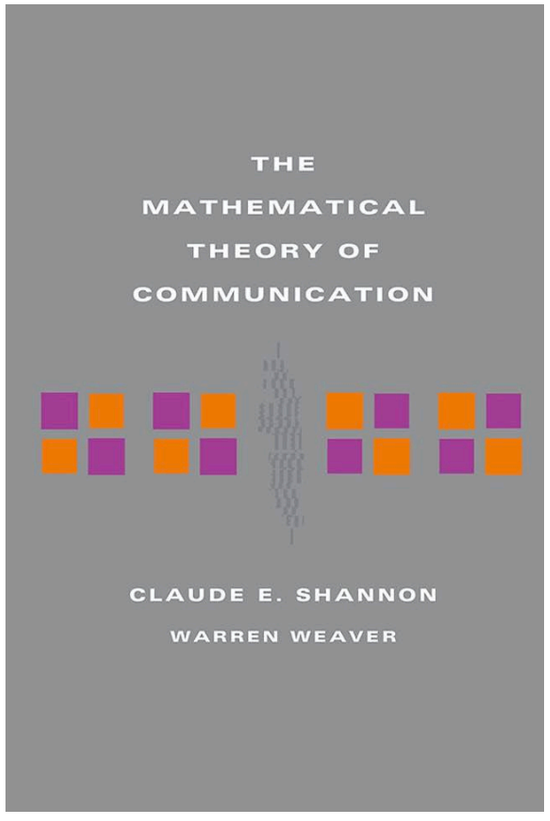
By contradiction.

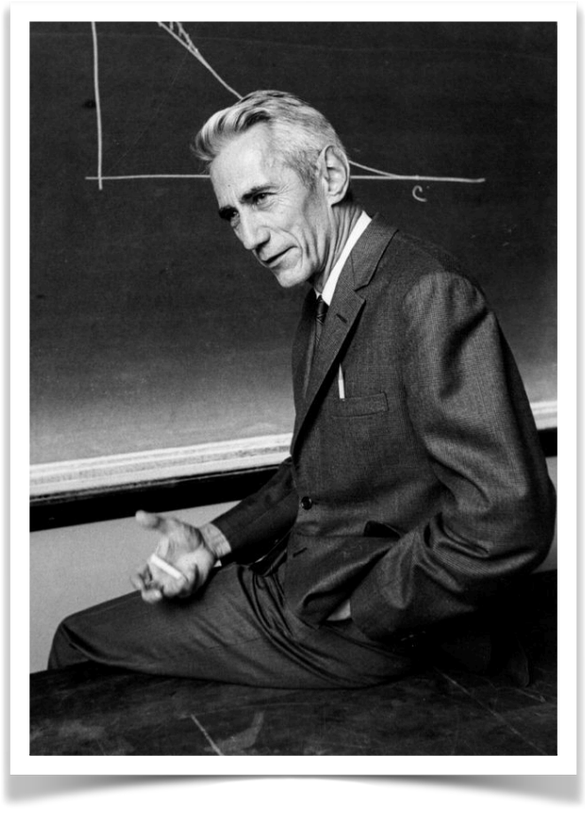
Information Theory

A Mathematical Theory of Communication

By C. E. SHANNON

Reprinted with corrections from *The Bell System Technical Journal,*
Vol. 27, pp. 379–423, 623–656, July, October, 1948.





Source Encoding

- Source coding protocol: Represent each element as a fixed binary code. Given a sequence of elements, just concatenate the codes.
  The target is to make the expected code length shorter.

- Consider sending the competition result of Chinese football team:
  Win: p=0.1 Draw: p=0.3 Lose: p=0.6

- Prefix-free coding:
  - No ambiguous for decoding
  - Apply shorter codes for high-probability elements can reduce the expected code length.
  - We can not make code length arbitrarily short: For prefix codes, Kraft's inequality holds:

Win	Draw	Lose
11	10	0

$$\sum_{i=1}^{N} 2^{-L_i} \leq 1$$

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

- A natural measure of information for an event with prob. p: $\log \frac{1}{p}$

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

- A natural measure of information for an event with prob. p: $\log \frac{1}{p}$
- The entropy of a random variable (or a prob. distribution) is the expected information contained:

$$H(x) = \sum_{i=1}^{N} p_i \log \frac{1}{p_i}$$

The Source Coding Theorem

- Apply shorter codes for high-probability elements can reduce the expected code length.

The longer the code, the more information included.
Low probability events have more information, it is "surprised" to receive them

- A natural measure of information for an event with prob. p: $\log \frac{1}{p}$
- The entropy of a random variable (or a prob. distribution) is the expected information contained:

$$H(x) = \sum_{i=1}^{N} p_i \log \frac{1}{p_i}$$

Source coding theorem:
Entropy is the (exact) lower bound for source coding length!

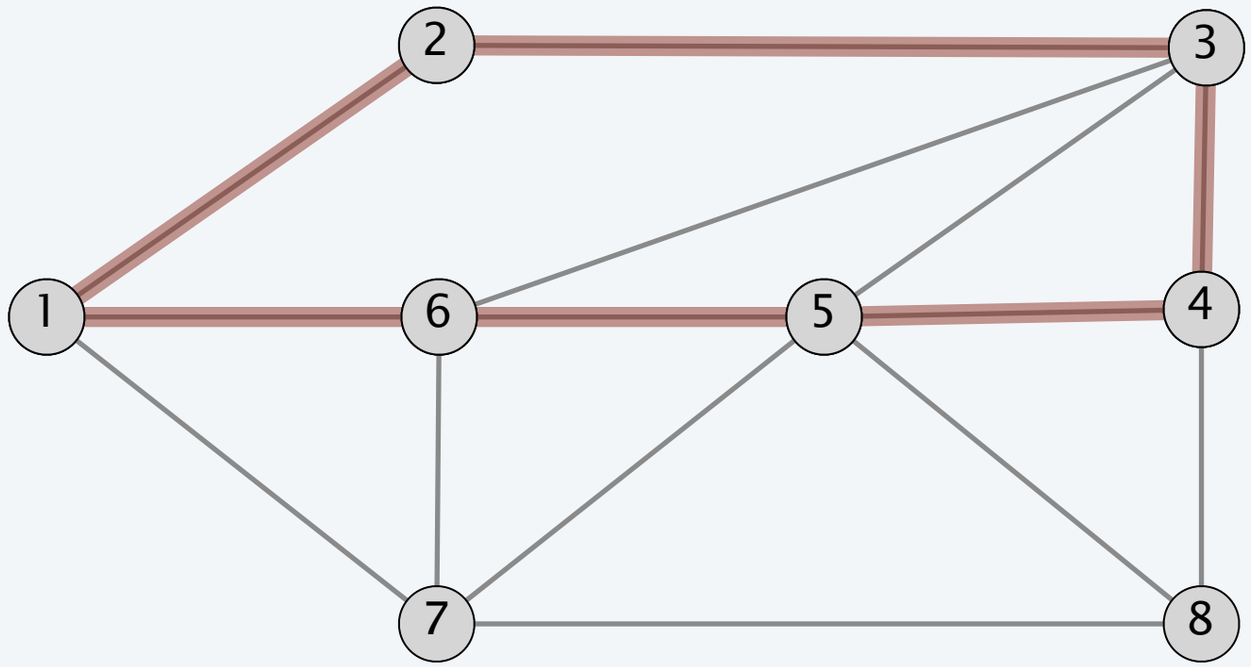
Greedy Algorithms

- Active selection (aka interval scheduling)
- Huffman coding
- Minimum spanning trees
- Take-home messages

Cycles

Def. A path is a sequence of edges which connects a sequence of nodes.

Def. A cycle is a path with no repeated nodes or edges other than the starting and ending nodes.



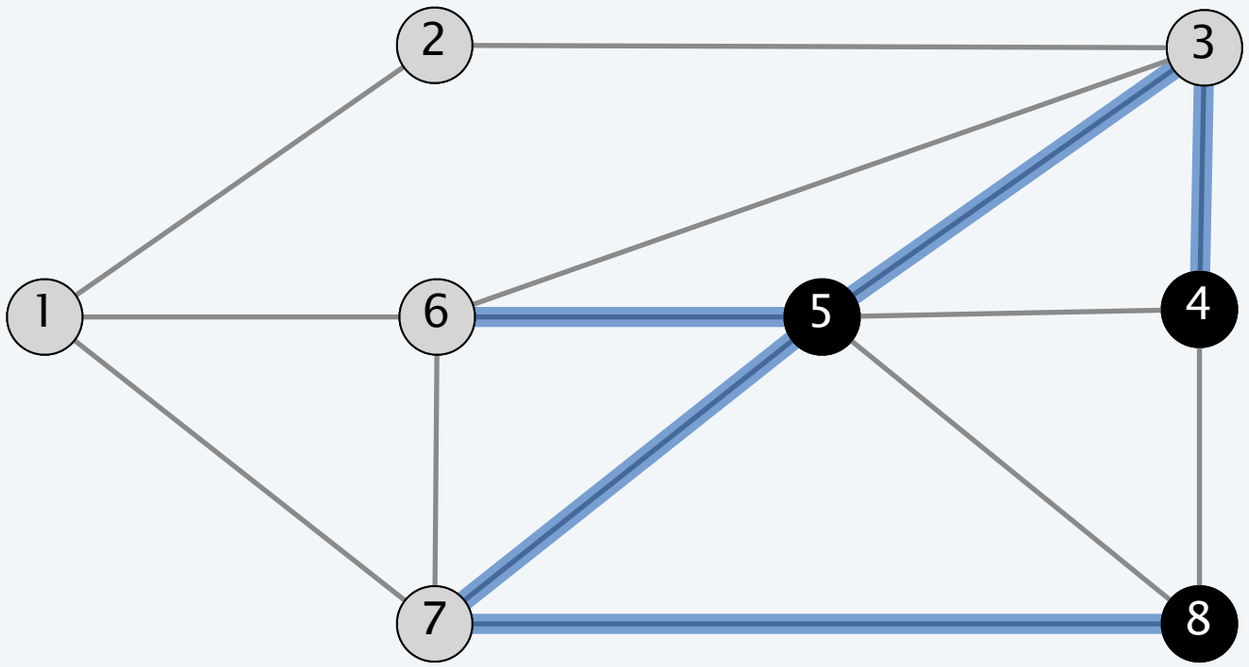
path P = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6) }

cycle C = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) }

Cuts

Def. A cut is a partition of the nodes into two nonempty subsets S and $V - S$.

Def. The cutset of a cut S is the set of edges with exactly one endpoint in S.



cut S = { 4, 5, 8 }

cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

Minimum spanning trees: quiz 1

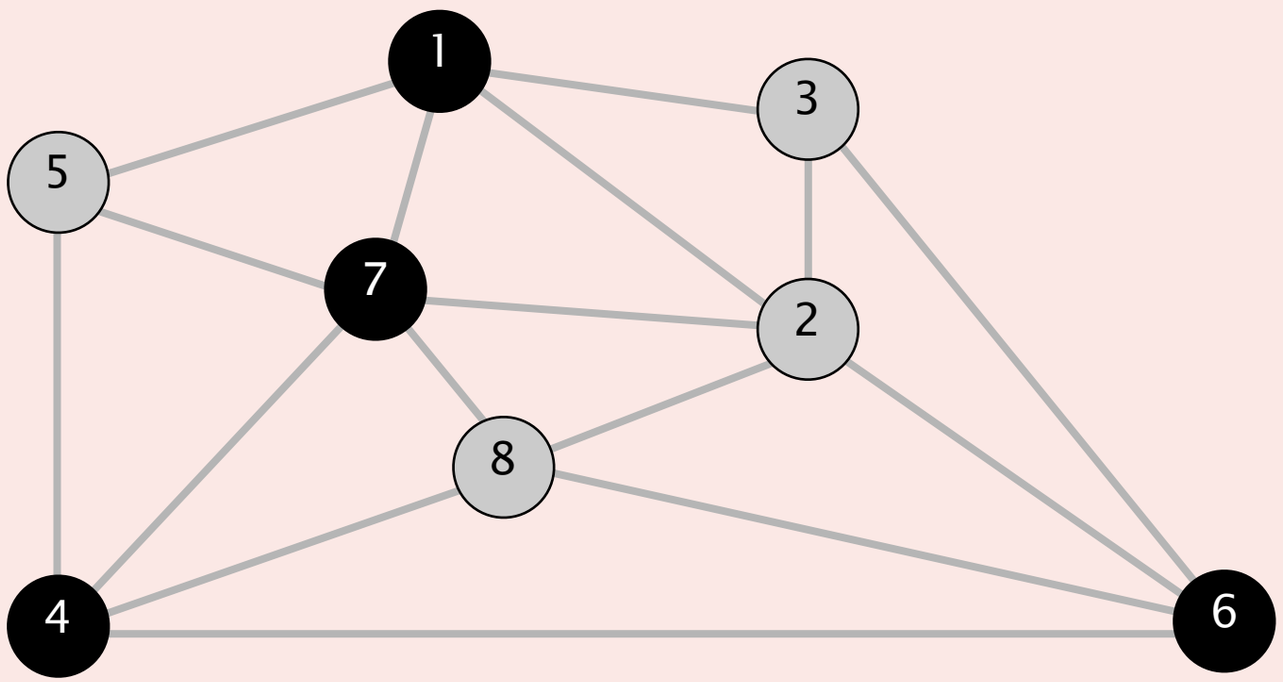
Consider the cut S = { 1, 4, 6, 7 }. Which edge is in the cutset of S?

A. S is not a cut (not connected)

B. 1–7

C. 5–7

D. 2–3



Minimum spanning trees: quiz 2

Let C be a cycle and let D be a cutset. How many edges do C and D have in common? Choose the best answer.

A. 0

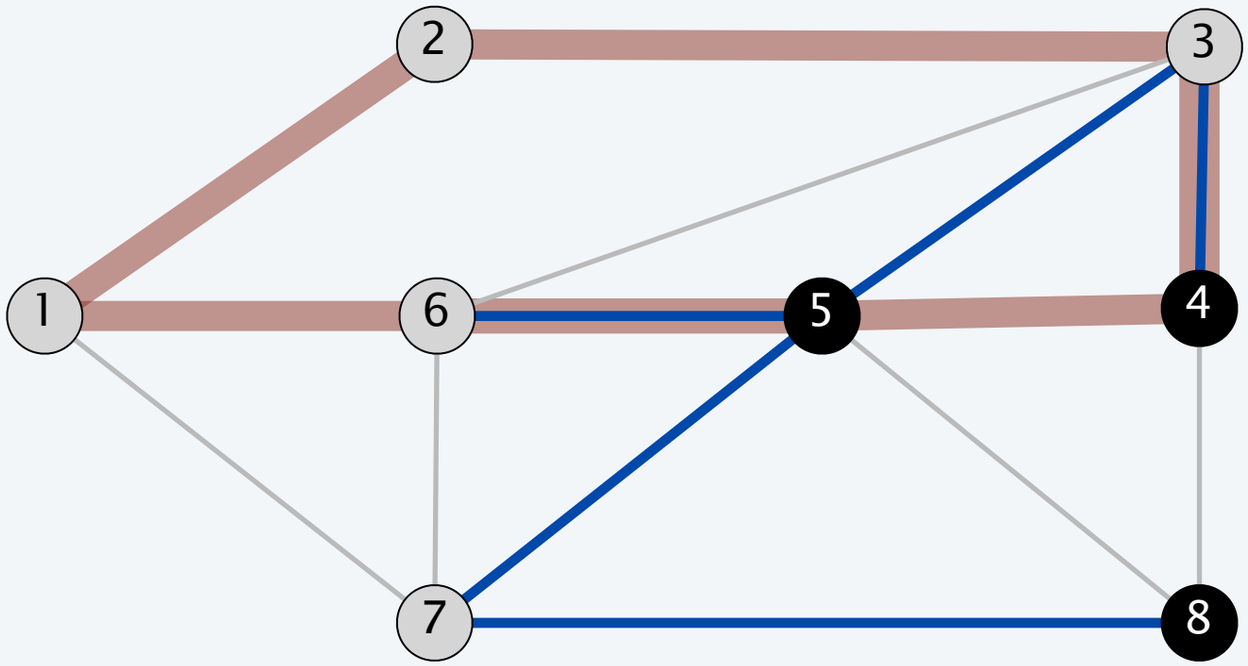
B. 2

C. not 1

D. an even number

Cycle–cut intersection

Proposition. A cycle and a cutset intersect in an even number of edges.



cycle C = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) }

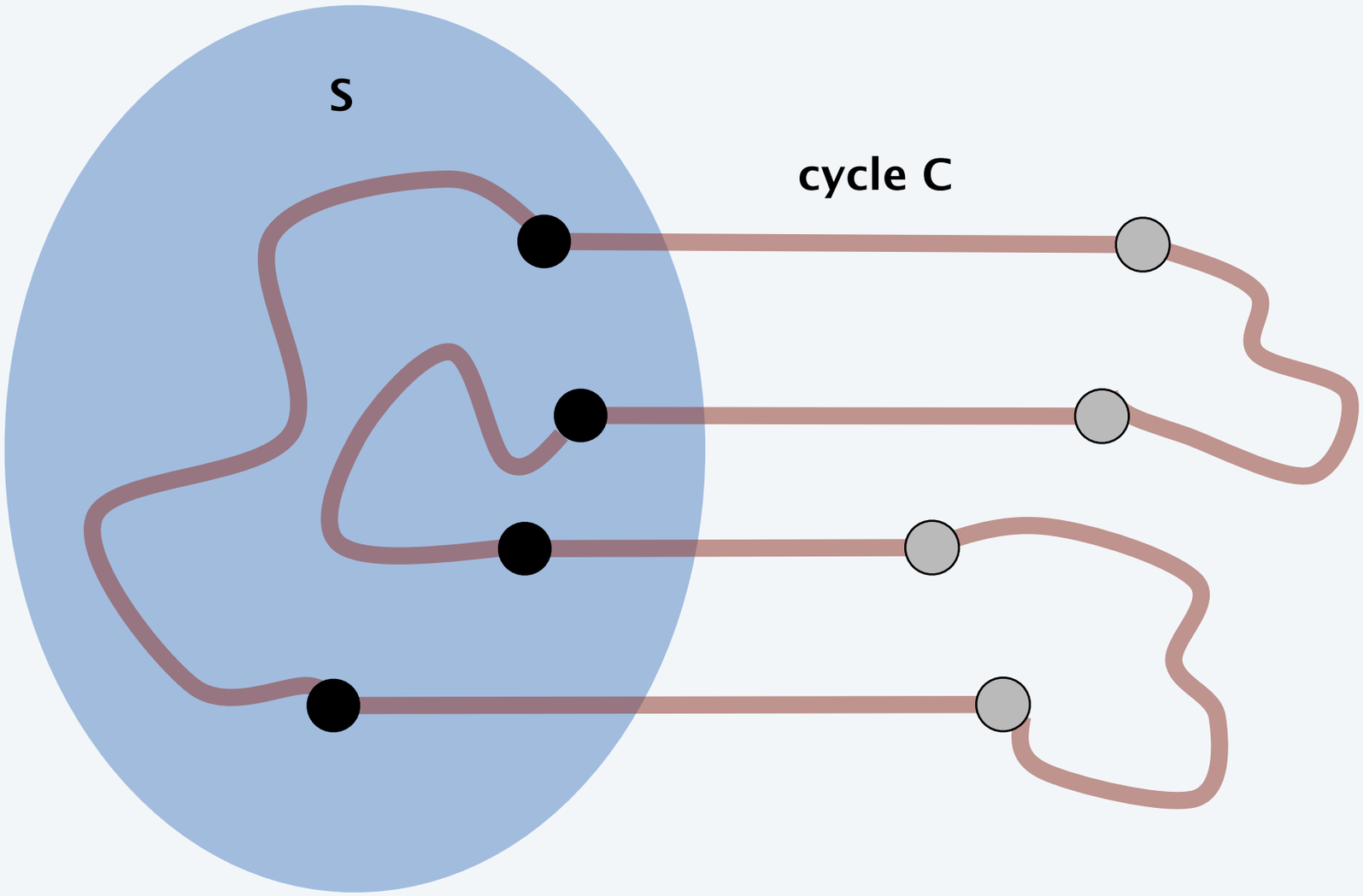
cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

intersection C ∩ D = { (3, 4), (5, 6) }

Cycle–cut intersection

Proposition. A cycle and a cutset intersect in an even number of edges.

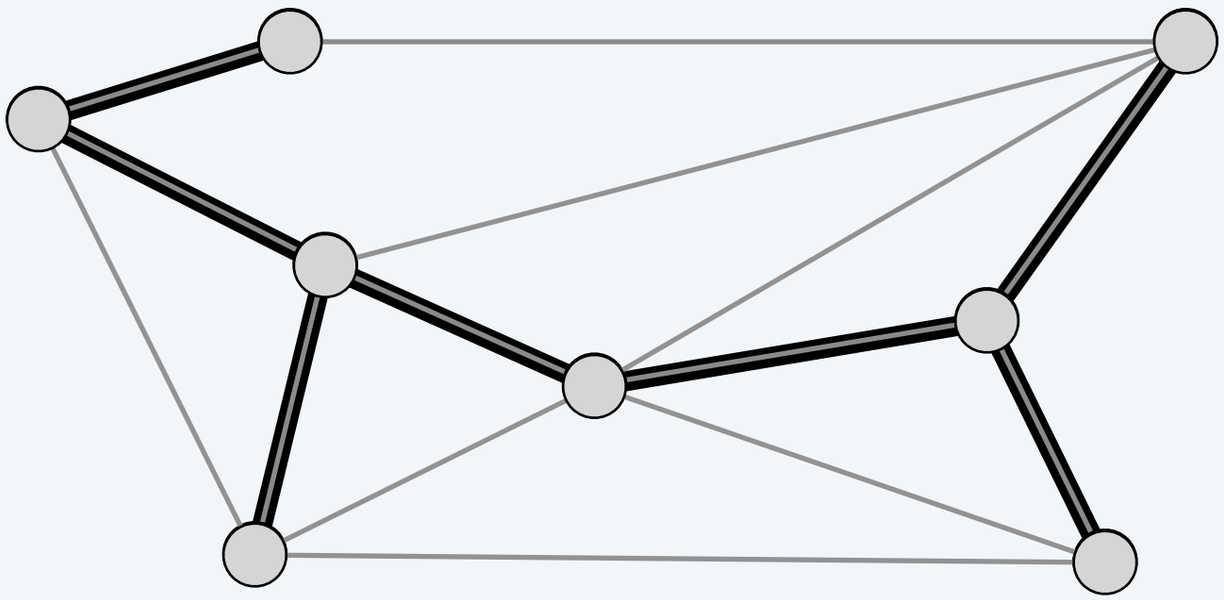
Pf. [by picture]



Spanning tree definition

Def. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$.

H is a spanning tree of G if H is both acyclic and connected.



graph G = (V, E)

spanning tree H = (V, T)

Minimum spanning trees: quiz 3

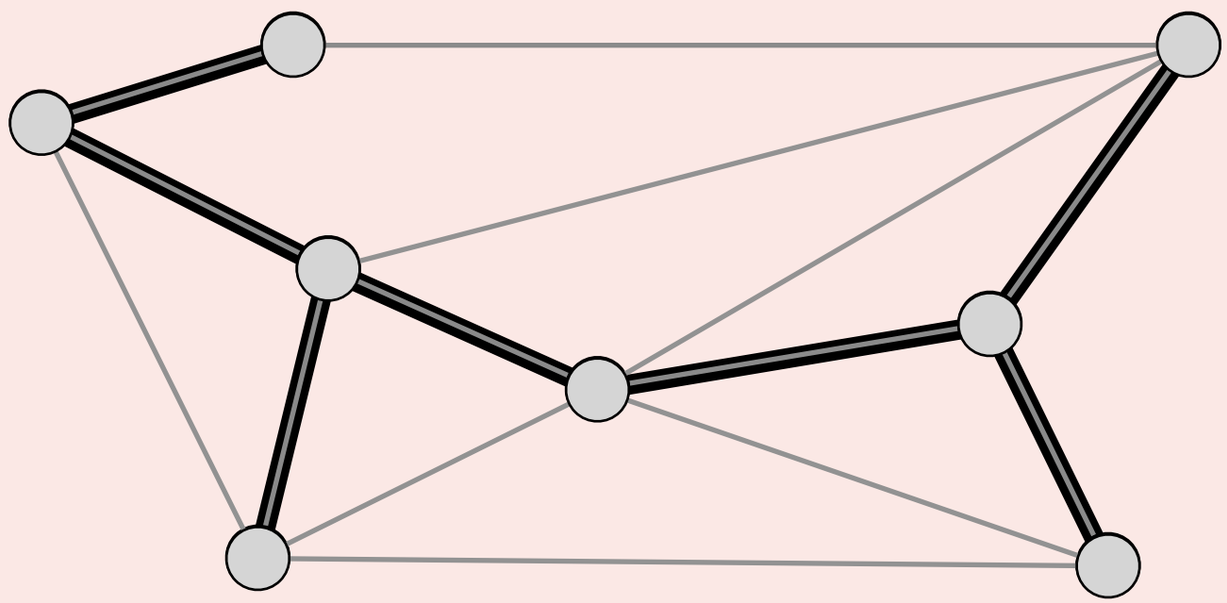
Which of the following properties are true for all spanning trees H?

A. Contains exactly $|V| - 1$ edges.

B. The removal of any edge disconnects it.

C. The addition of any edge creates a cycle.

D. All of the above.



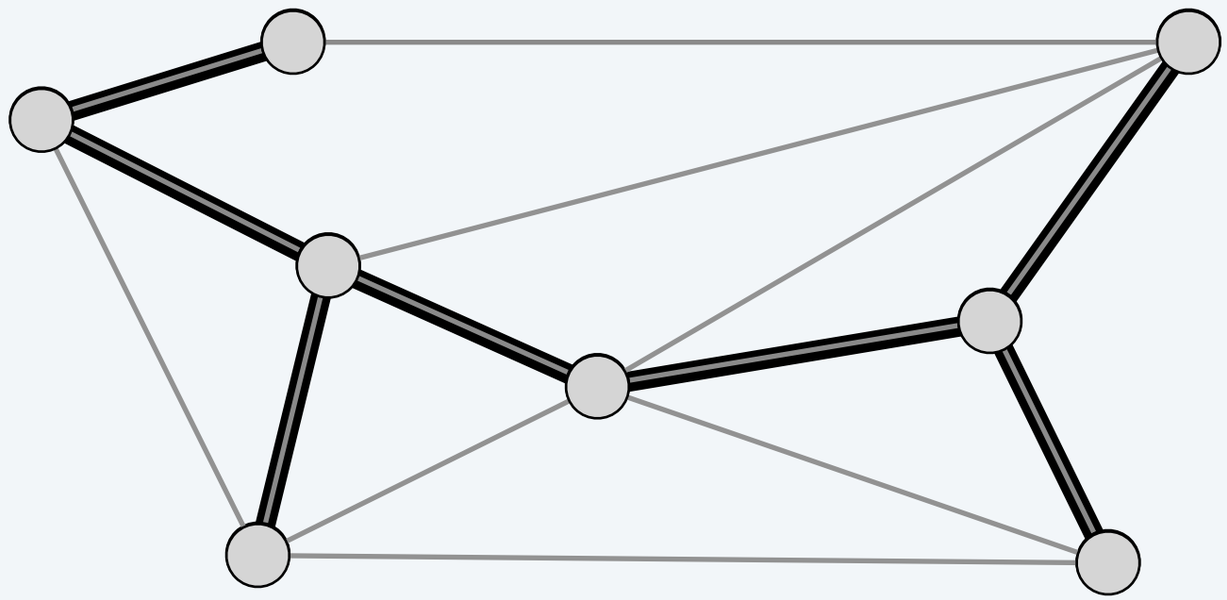
graph G = (V, E)

spanning tree H = (V, T)

Spanning tree properties

Proposition. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$.
Then, the following are equivalent:

- H is a spanning tree of G.
- H is acyclic and connected.
- H is connected and has $|V| - 1$ edges.
- H is acyclic and has $|V| - 1$ edges.
- H is minimally connected: removal of any edge disconnects it.
- H is maximally acyclic: addition of any edge creates a cycle.



graph G = (V, E)

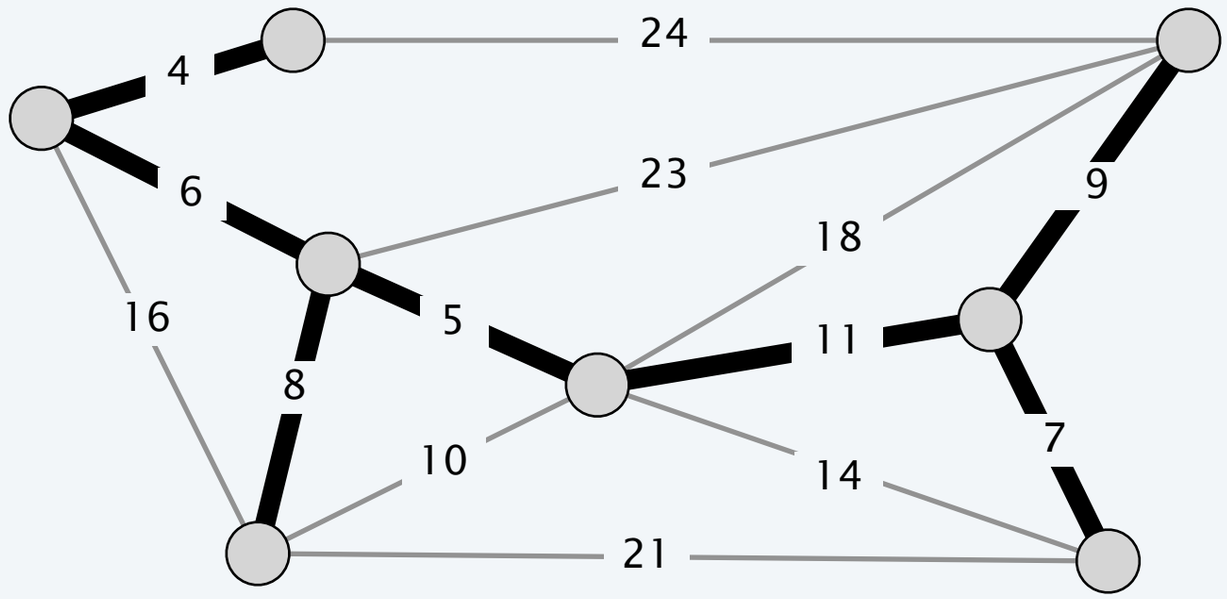
spanning tree H = (V, T)

A tree containing a cycle

https://maps.roadtrippers.com/places/46955/photos/374771356

Minimum spanning tree (MST)

Def. Given a connected, undirected graph $G = (V, E)$ with edge costs c_e, a minimum spanning tree (V, T) is a spanning tree of G such that the sum of the edge costs in T is minimized.



MST cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Cayley's theorem. The complete graph on n nodes has n^{n-2} spanning trees.

can't solve by brute force

Minimum spanning trees: quiz 4

Suppose that you change the cost of every edge in G as follows. For which is every MST in G an MST in G′ (and vice versa)? Assume c(e) > 0 for each e.

A. $c'(e) = c(e) + 17.$

B. $c'(e) = 17 \times c(e).$

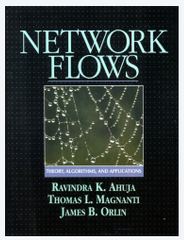
C. $c'(e) = \log_{17} c(e).$

D. All of the above.

Applications

MST is fundamental problem with diverse applications.

- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Model locality of particle interactions in turbulent fluid flows.
- Reducing data storage in sequencing amino acids in a protein.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).

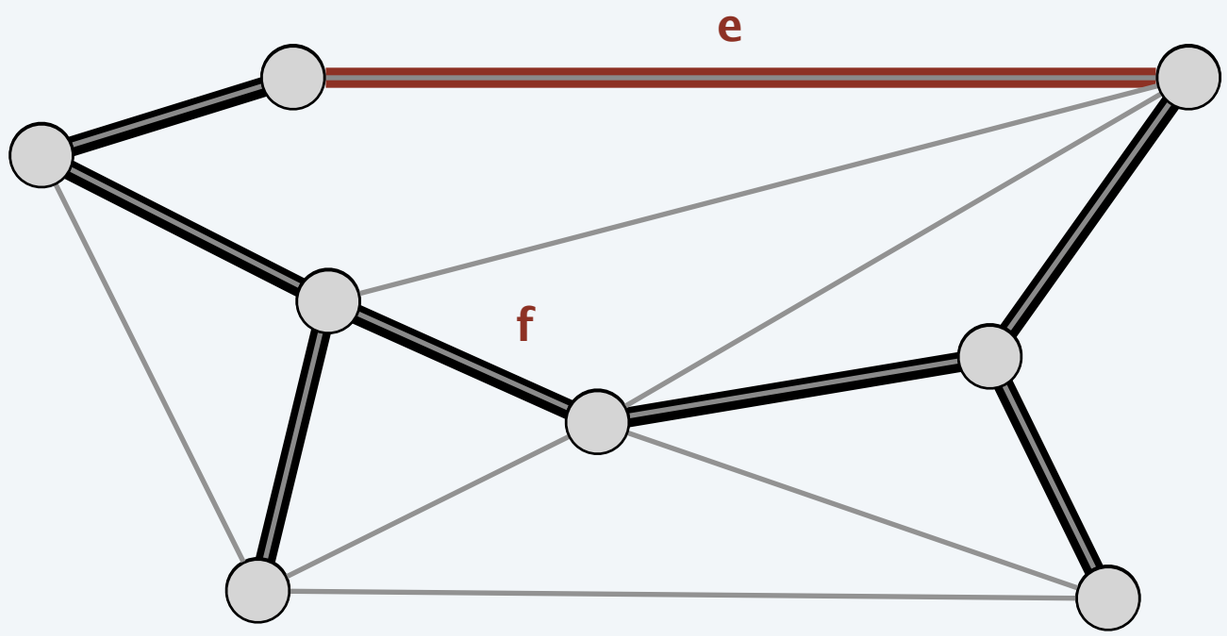


Network Flows: Theory, Algorithms, and Applications, by Ahuja, Magnanti, and Orlin, Prentice Hall, 1993.

Fundamental cycle

Fundamental cycle. Let $H = (V, T)$ be a spanning tree of $G = (V, E)$.

- For any non tree-edge $e \in E$: $T \cup \{ e \}$ contains a unique cycle, say C.
- For any edge $f \in C$: $(V, T \cup \{ e \} - \{ f \})$ is a spanning tree.



graph G = (V, E)

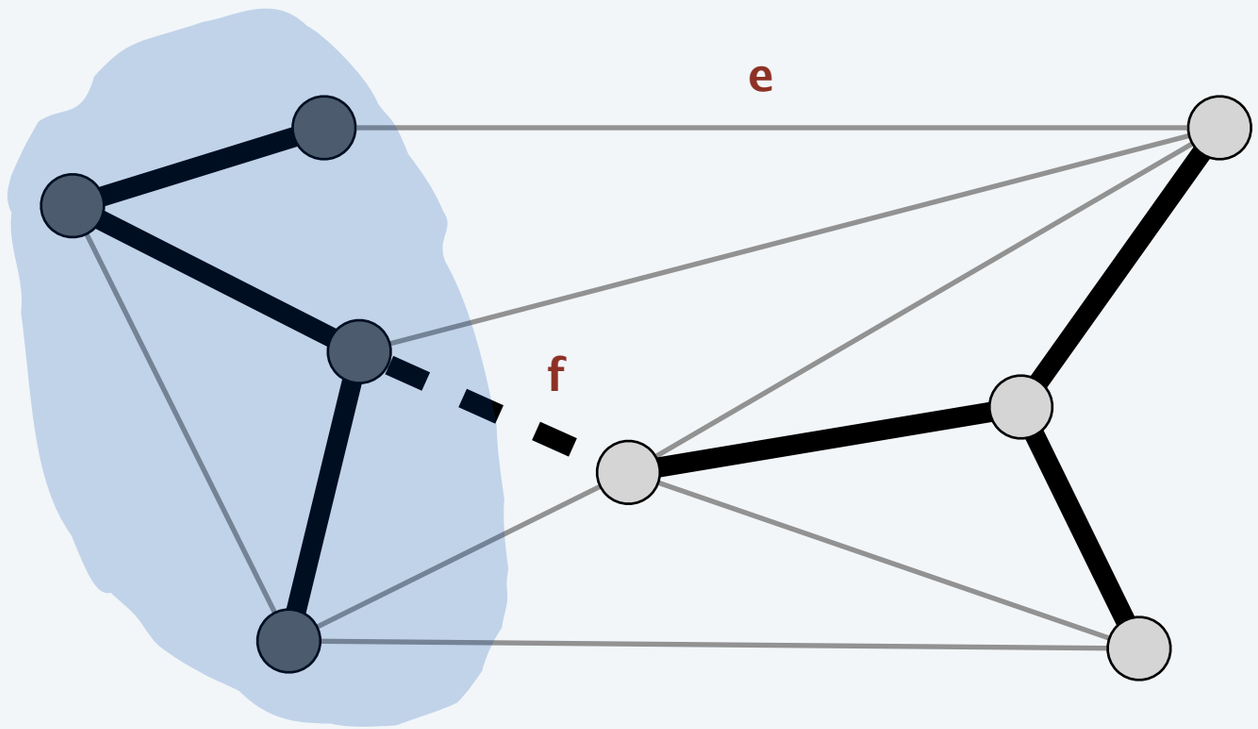
spanning tree H = (V, T)

Observation. If $c_e < c_f$, then (V, T) is not an MST.

Fundamental cutset

Fundamental cutset. Let $H = (V, T)$ be a spanning tree of $G = (V, E)$.

- For any tree edge $f \in T$: $(V,\ T - \{ f \})$ has two connected components.
- Let D denote corresponding cutset.
- For any edge $e \in D$: $(V,\ T - \{ f \} \cup \{ e \})$ is a spanning tree.



graph G = (V, E)

spanning tree H = (V, T)

Observation. If $c_e < c_f$, then (V, T) is not an MST.

The greedy algorithm

Red rule.

- Let C be a cycle with no red edges.
- Select an uncolored edge of C of max cost and color it red.

Blue rule.

- Let D be a cutset with no blue edges.
- Select an uncolored edge in D of min cost and color it blue.

Greedy algorithm.

- Apply the red and blue rules (nondeterministically!) until all edges are colored. The blue edges form an MST.
- Note: can stop once $n - 1$ edges colored blue.

Greedy algorithm: proof of correctness

Color invariant. There exists an MST (V, T^*) containing every blue edge and no red edge.

Pf. [by induction on number of iterations]

Base case. No edges colored $\Rightarrow$ every MST satisfies invariant.

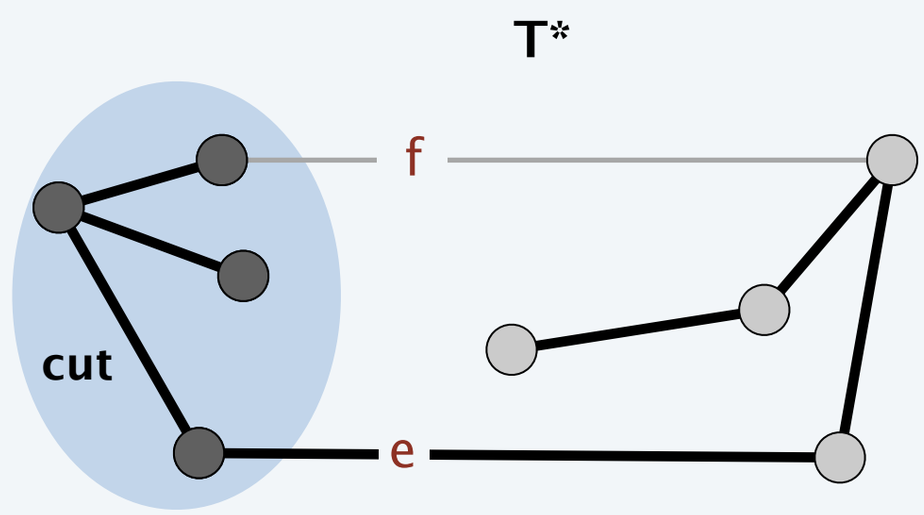
Greedy algorithm: proof of correctness

Color invariant. There exists an MST (V, T^*) containing every blue edge and no red edge.

Pf. [by induction on number of iterations]

Induction step (blue rule). Suppose color invariant true before blue rule.

- let D be chosen cutset, and let f be edge colored blue.
- if $f \in T^*$, then T^* still satisfies invariant.
- Otherwise, consider fundamental cycle C by adding f to T^*.
- let $e \in C$ be another edge in D.
- e is uncolored and $c_e \geq c_f$ since
  - $e \in T^* \Rightarrow e$ not red
  - blue rule $\Rightarrow e$ not blue and $c_e \geq c_f$
- Thus, $T^* \cup \{ f \} - \{ e \}$ satisfies invariant.



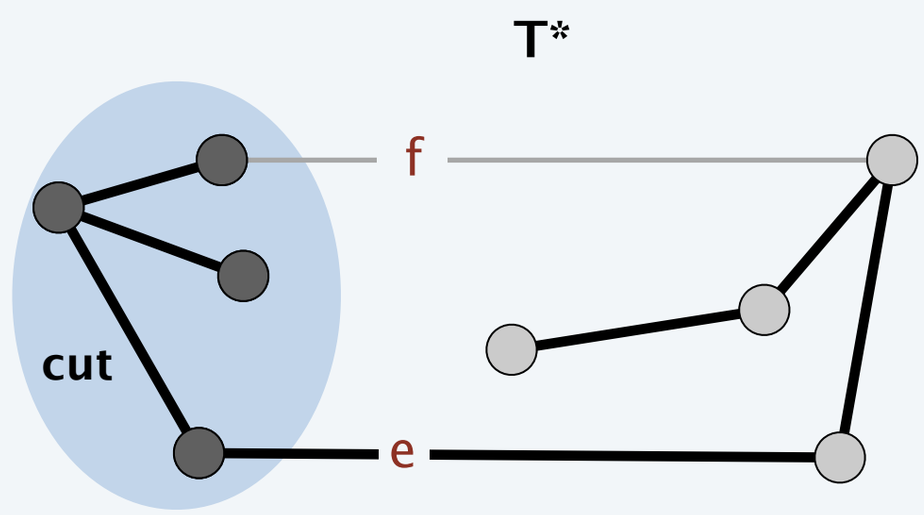
Greedy algorithm: proof of correctness

Color invariant. There exists an MST (V, T^*) containing every blue edge and no red edge.

Pf. [by induction on number of iterations]

Induction step (red rule). Suppose color invariant true before red rule.

- let C be chosen cycle, and let e be edge colored red.
- if $e \notin T^*$, then T^* still satisfies invariant.
- Otherwise, consider fundamental cutset D by deleting e from T^*.
- let $f \in D$ be another edge in C.
- f is uncolored and $c_e \geq c_f$ since
  - $f \notin T^* \Rightarrow f$ not blue
  - red rule $\Rightarrow f$ not red and $c_e \geq c_f$
- Thus, $T^* \cup \{ f \} - \{ e \}$ satisfies invariant. ▪



Greedy algorithm: proof of correctness

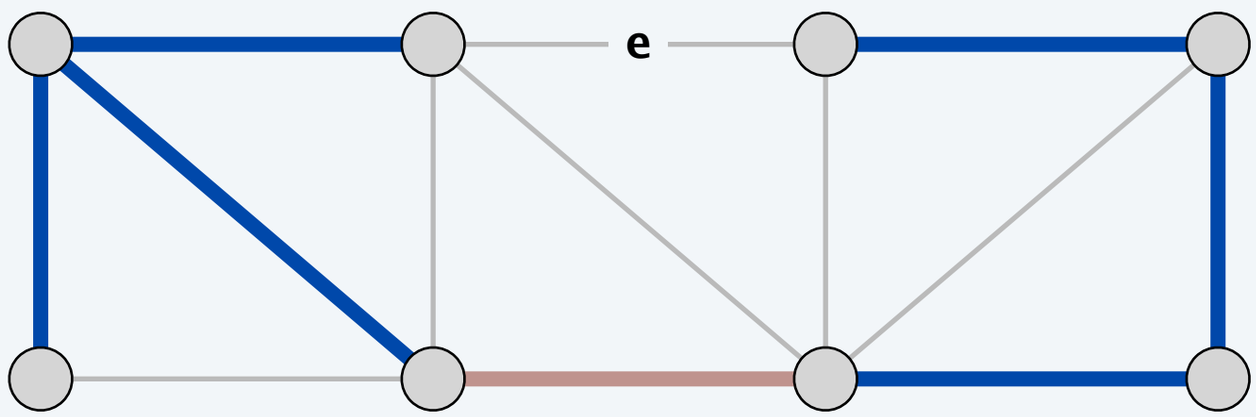
Theorem. The greedy algorithm terminates. Blue edges form an MST.

Pf. We need to show that either the red or blue rule (or both) applies.

- Suppose edge e is left uncolored.
- Blue edges form a forest.
- Case 1: both endpoints of e are in same blue tree.

  $\Rightarrow$ apply red rule to cycle formed by adding e to blue forest.
- Case 2: both endpoints of e are in different blue trees.

  $\Rightarrow$ apply blue rule to cutset induced by either of two blue trees. ▪



Case 2

Prim's algorithm

Initialize $S = \{ s \}$ for any node s, $T = \emptyset$.

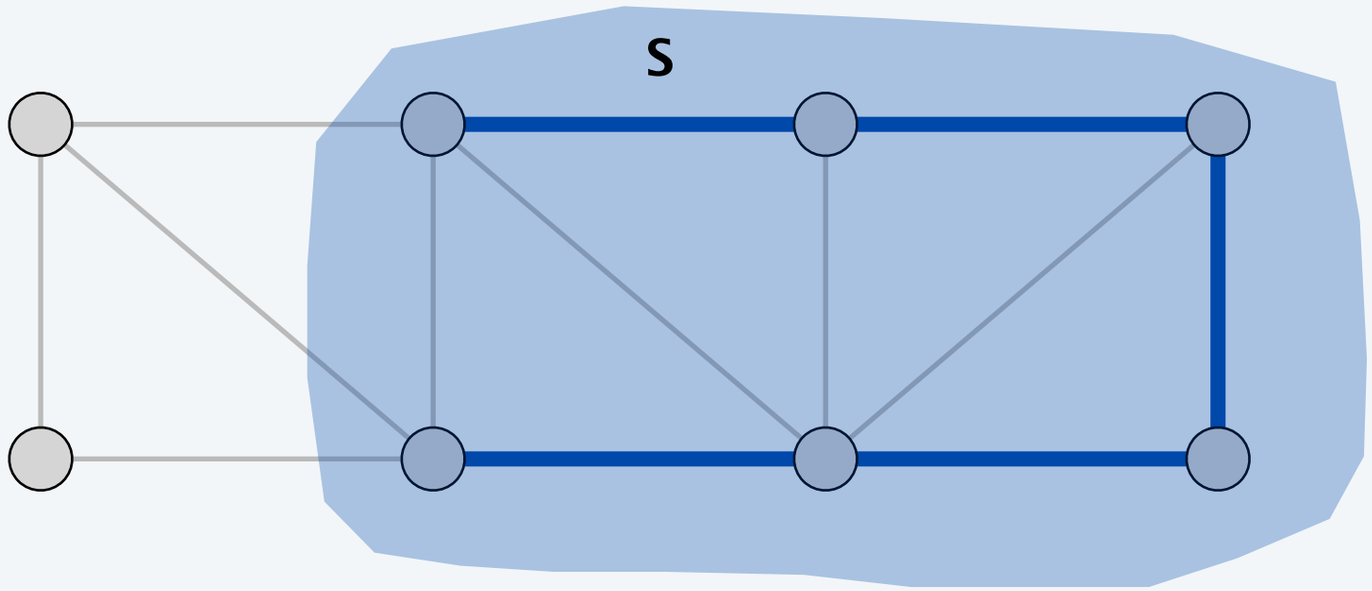
Repeat $n - 1$ times:

- Add to T a min-cost edge with exactly one endpoint in S.
- Add the other endpoint to S.

by construction, edges in cutset are uncolored

Theorem. Prim's algorithm computes an MST.

Pf. Special case of greedy algorithm (blue rule repeatedly applied to S). ▪



Prim's algorithm: implementation

Theorem. Prim's algorithm can be implemented to run in $O(m \log n)$ time.

Pf. Implementation almost identical to Dijkstra's algorithm.

PRIM (V, E, c)

$S \leftarrow \emptyset$, $T \leftarrow \emptyset$.

$s \leftarrow$ any node in V.

FOREACH $v \neq s$: $\pi[v] \leftarrow \infty$, $pred[v] \leftarrow null$; $\pi[s] \leftarrow 0$.

Create an empty priority queue pq.

FOREACH $v \in V$: INSERT$(pq, v, \pi[v])$.

WHILE (IS-NOT-EMPTY(pq))

  $u \leftarrow$ DEL-MIN(pq).

  $S \leftarrow S \cup \{ u \}$, $T \leftarrow T \cup \{ pred[u] \}$.

  FOREACH edge $e = (u, v) \in E$ with $v \notin S$:

    IF $(c_e < \pi[v])$

      DECREASE-KEY(pq, v, c_e).

      $\pi[v] \leftarrow c_e$; $pred[v] \leftarrow e$.

$\pi[v]$ = cost of cheapest known edge between v and S

Kruskal's algorithm

Consider edges in ascending order of cost:

- Add to tree unless it would create a cycle.

Theorem. Kruskal's algorithm computes an MST.

Pf. Special case of greedy algorithm.

- Case 1: both endpoints of e in same blue tree.

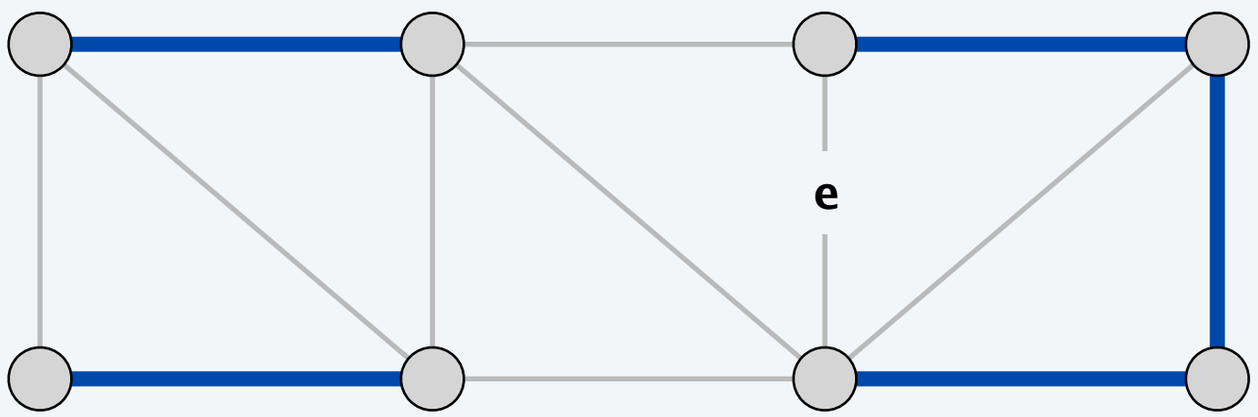
  $\Rightarrow$ color e red by applying red rule to unique cycle.

all other edges in cycle are blue

- Case 2: both endpoints of e in different blue trees.

  $\Rightarrow$ color e blue by applying blue rule to cutset defined by either tree. ▪

no edge in cutset has smaller cost
(since Kruskal chose it first)



Kruskal's algorithm: implementation

Theorem. Kruskal's algorithm can be implemented to run in $O(m \log m)$ time.

- Sort edges by cost.
- Use union–find data structure to dynamically maintain connected components.

KRUSKAL (V, E, c)

SORT m edges by cost and renumber so that $c(e_1) \le c(e_2) \le \ldots \le c(e_m)$.

$T \leftarrow \emptyset$.

FOREACH $v \in V$: MAKE-SET(v).

FOR $i = 1$ TO m

  $(u, v) \leftarrow e_i$.

  IF (FIND-SET$(u) \neq$ FIND-SET(v)) ← are u and v in same component?

    $T \leftarrow T \cup \{ e_i \}$.

    UNION(u, v). ← make u and v in same component

RETURN T.

Reverse-delete algorithm

Start with all edges in T and consider them in descending order of cost:

- Delete edge from T unless it would disconnect T.

Theorem. The reverse-delete algorithm computes an MST.

Pf. Special case of greedy algorithm.

- Case 1. [deleting edge e does not disconnect T]

  $\Rightarrow$ apply red rule to cycle C formed by adding e to another path in T between its two endpoints

  no edge in C is more expensive
  (it would have already been considered and deleted)

- Case 2. [deleting edge e disconnects T]

  $\Rightarrow$ apply blue rule to cutset D induced by either component ▪

  e is the only remaining edge in the cutset
  (all other edges in D must have been colored red / deleted)

Fact. [Thorup 2000] Can be implemented to run in $O(m \log n (\log \log n)^3)$ time.

Review: the greedy MST algorithm

Red rule.

- Let C be a cycle with no red edges.
- Select an uncolored edge of C of max cost and color it red.

Blue rule.

- Let D be a cutset with no blue edges.
- Select an uncolored edge in D of min cost and color it blue.

Greedy algorithm.

- Apply the red and blue rules (nondeterministically!) until all edges are colored. The blue edges form an MST.
- Note: can stop once $n - 1$ edges colored blue.

Theorem. The greedy algorithm is correct.

Special cases. Prim, Kruskal, reverse-delete, ...

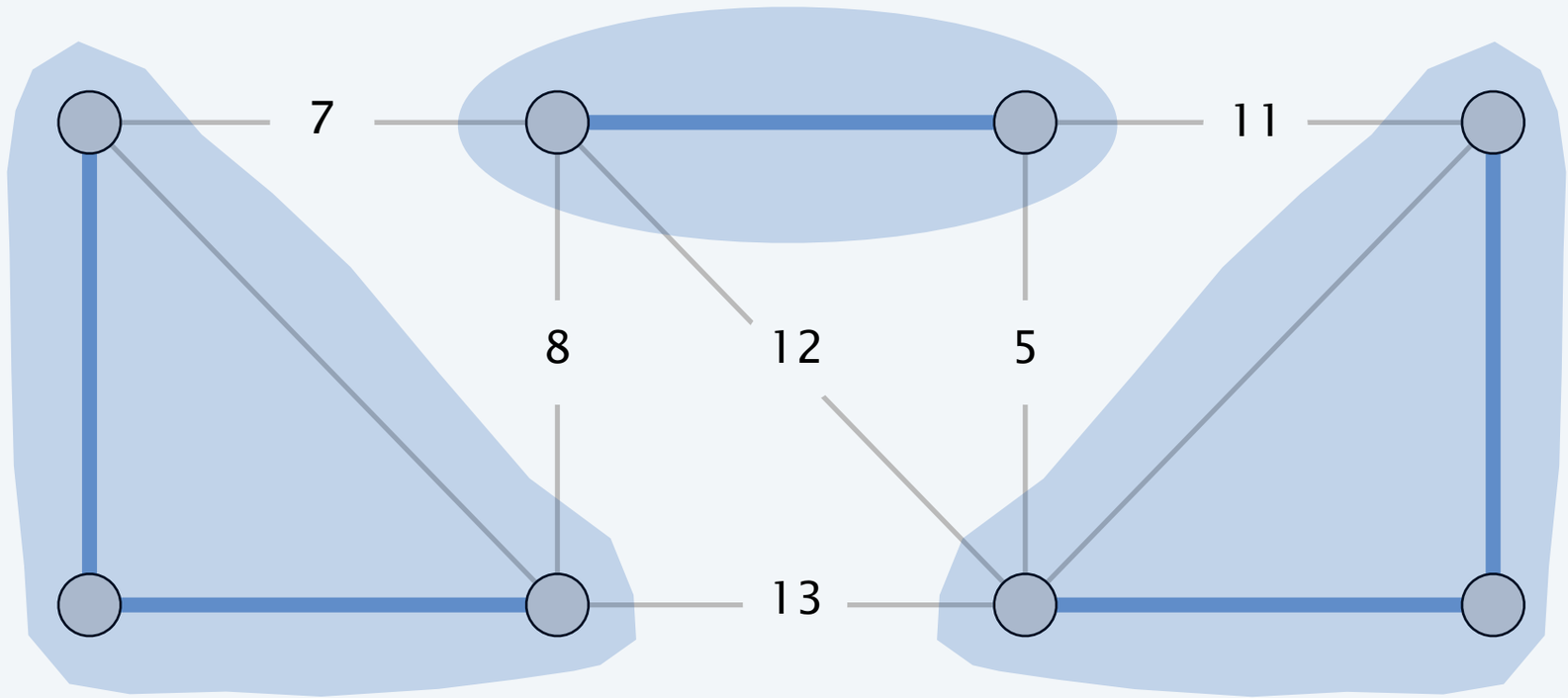
Borůvka's algorithm

Repeat until only one tree.

- Apply blue rule to cutset corresponding to each blue tree.
- Color all selected edges blue.

Theorem. Borůvka's algorithm computes the MST. ← assume edge costs are distinct

Pf. Special case of greedy algorithm (repeatedly apply blue rule). ▪

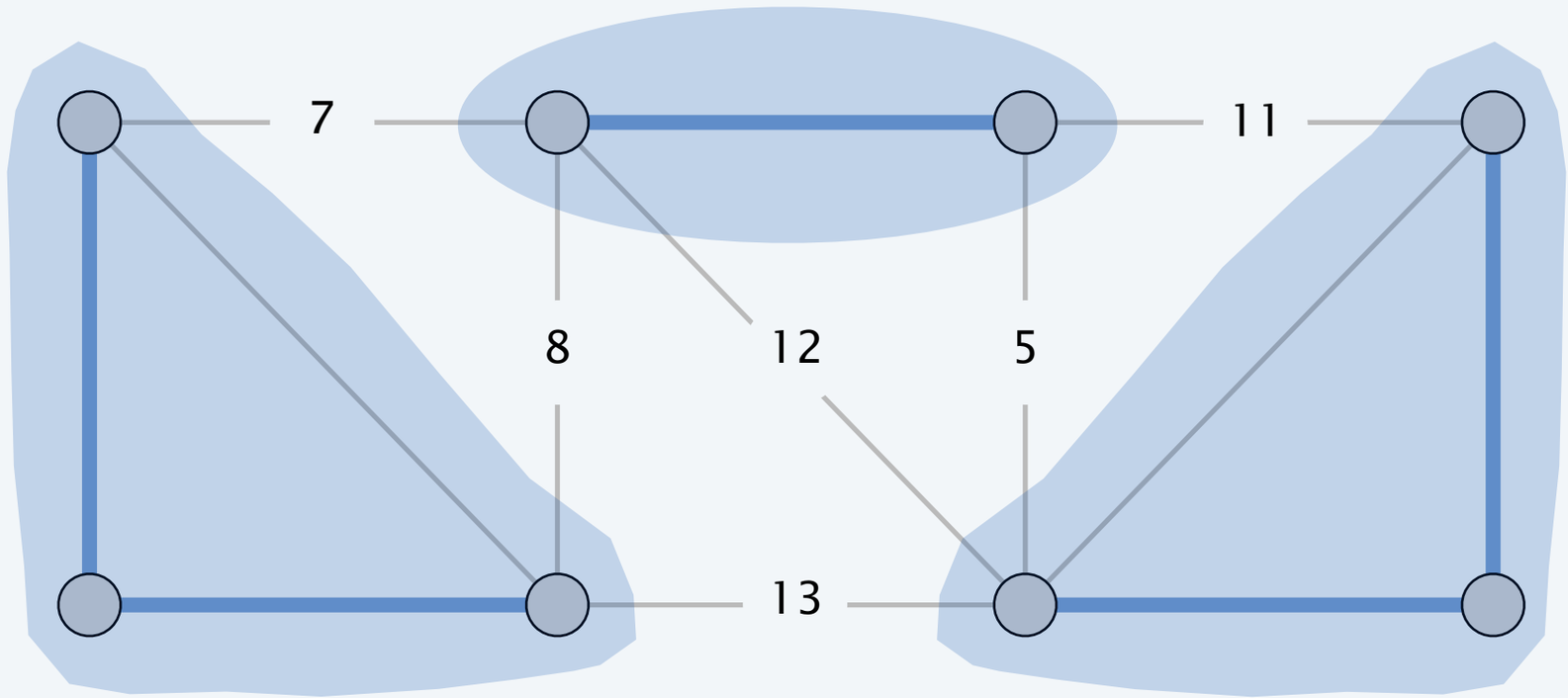


Borůvka's algorithm: implementation

Theorem. Borůvka's algorithm can be implemented to run in $O(m \log n)$ time.

Pf.

- To implement a phase in $O(m)$ time:
  - compute connected components of blue edges
  - for each edge $(u, v) \in E$, check if u and v are in different components; if so, update each component's best edge in cutset
- $\leq \log_2 n$ phases since each phase (at least) halves total # components. ▪

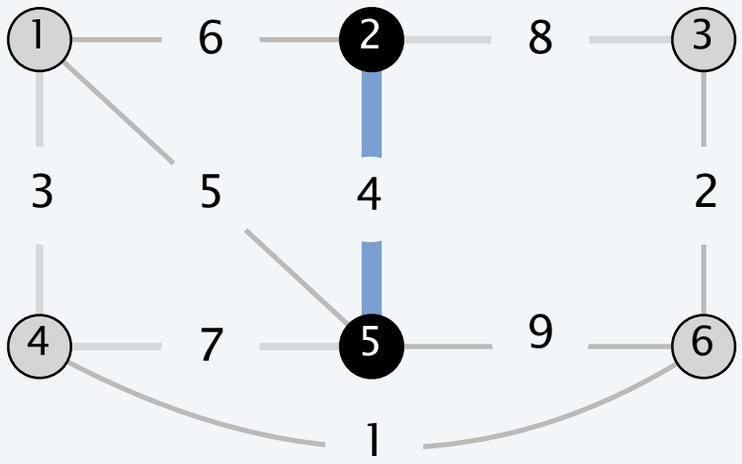


Borůvka's algorithm: implementation

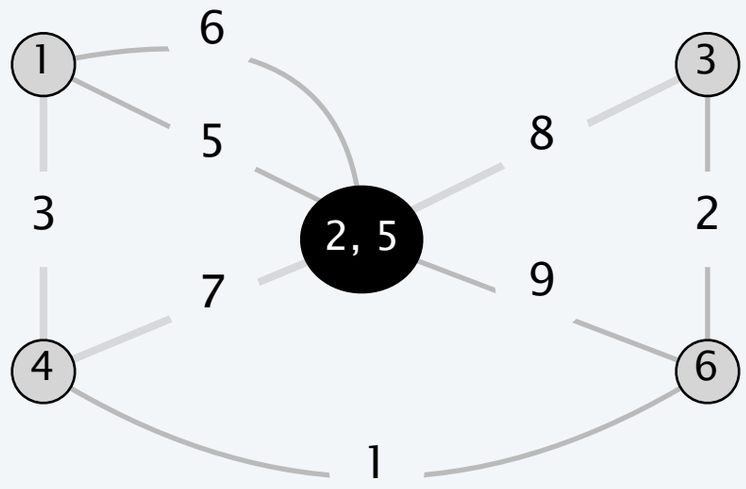
Contraction version.

- After each phase, contract each blue tree to a single supernode.
- Delete self-loops and parallel edges (keeping only cheapest one).
- Borůvka phase becomes: take cheapest edge incident to each node.

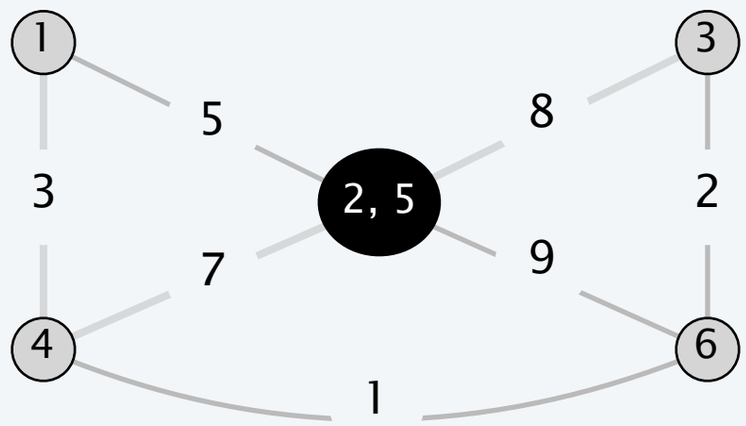
graph G



contract edge 2–5



delete self-loops and parallel edges

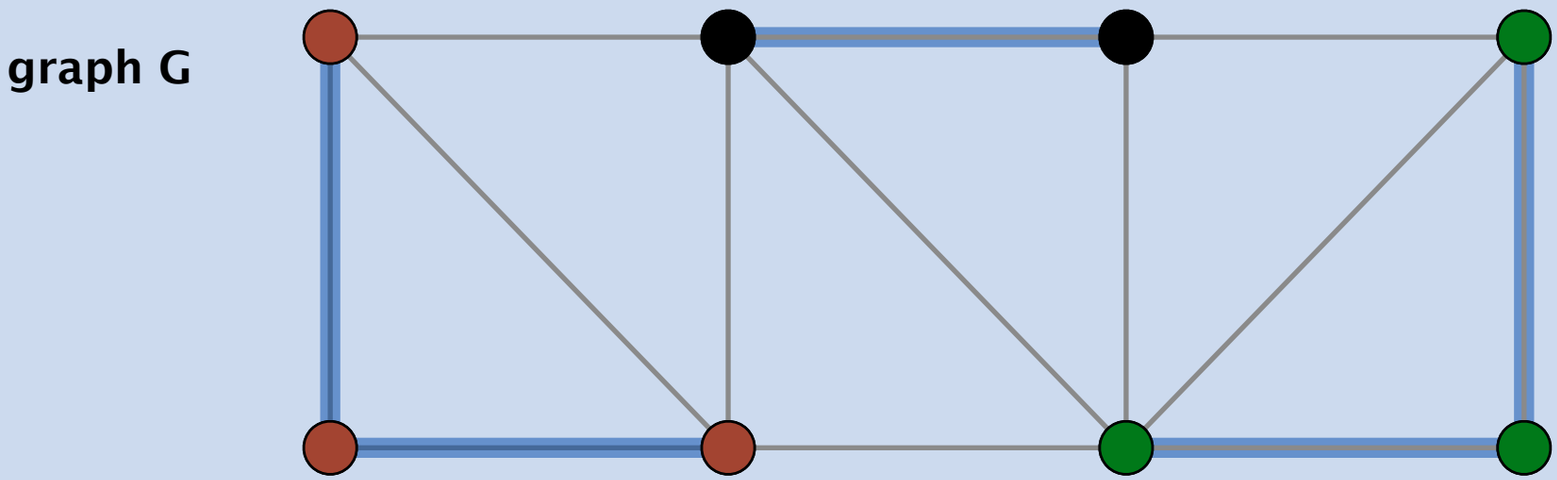


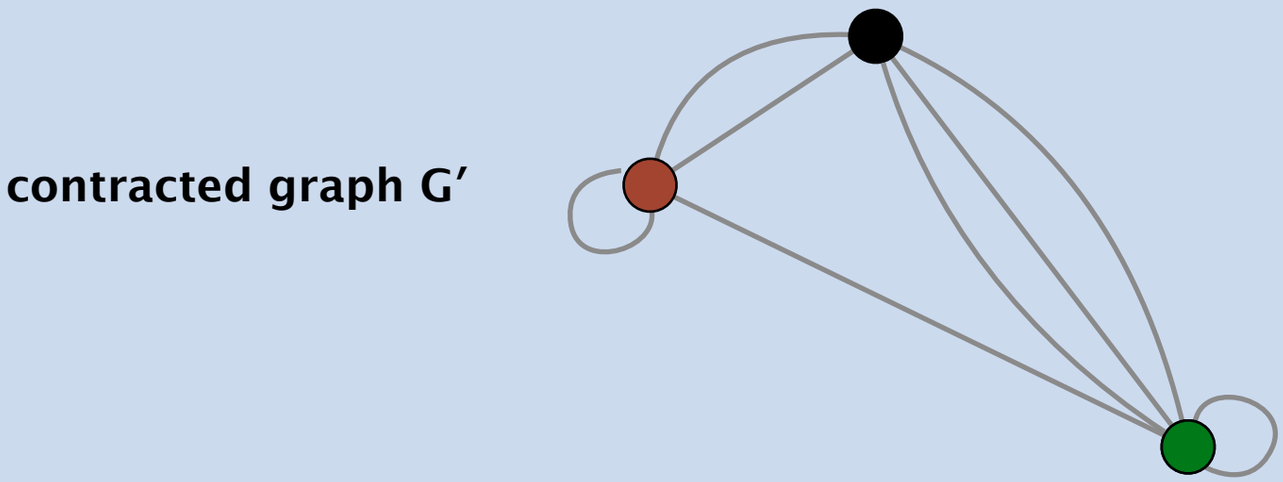
Q. How to contract a set of edges?

CONTRACT A SET OF EDGES

Problem. Given a graph $G = (V, E)$ and a set of edges F, contract all edges in F, removing any self-loops or parallel edges.

Goal. $O(m + n)$ time.





CONTRACT A SET OF EDGES

Problem. Given a graph $G = (V, E)$ and a set of edges F, contract all edges in F, removing any self-loops or parallel edges.

Solution.

- Compute the n' connected components in (V, F).
- Suppose $id[u] = i$ means node u is in connected component i.
- The contracted graph G' has n' nodes.
- For each edge u–$v \in E$, add an edge i–j to G', where $i = id[u]$ and $j = id[v]$.

Removing self loops. Easy.

Removing parallel edges.

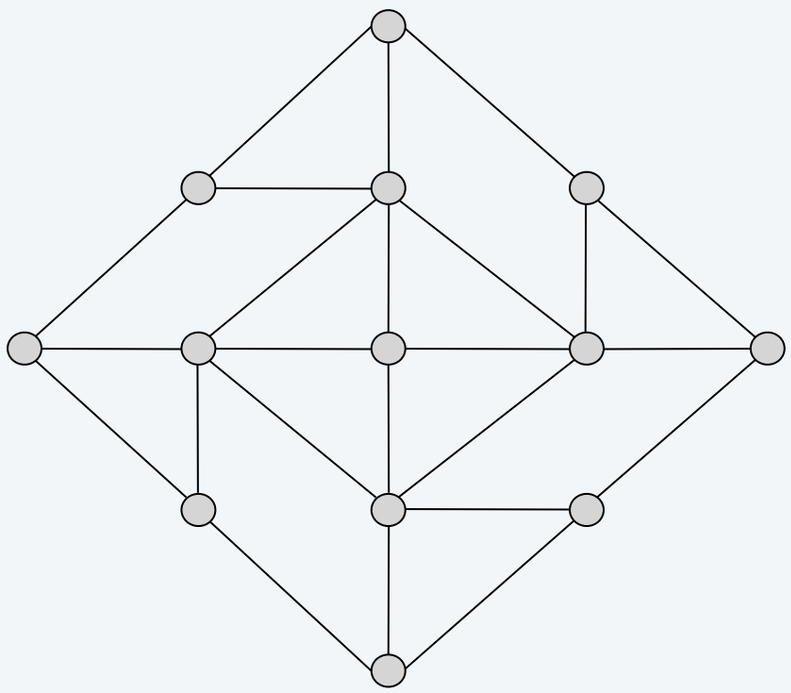
- Create a list of edges i–j with the convention that $i < j$.
- Sort the edges lexicographically via LSD radix sort.
- Add the edges to the graph G', removing parallel edges.

Borůvka's algorithm on planar graphs

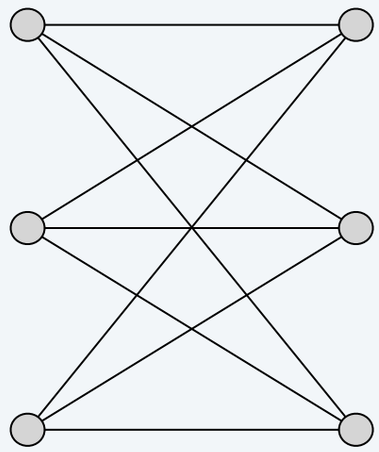
Theorem. Borůvka's algorithm (contraction version) can be implemented to run in $O(n)$ time on planar graphs.

Pf.

- Each Borůvka phase takes $O(n)$ time:
  - Fact 1: $m \leq 3n$ for simple planar graphs.
  - Fact 2: planar graphs remains planar after edge contractions/deletions.
- Number of nodes (at least) halves in each phase.
- Thus, overall running time $\leq cn + cn/2 + cn/4 + cn/8 + \ldots = O(n)$. ▪



planar



$K_{3,3}$ not planar

A hybrid algorithm

Borůvka–Prim algorithm.

- Run Borůvka (contraction version) for $\log_2 \log_2 n$ phases.
- Run Prim on resulting, contracted graph.

Theorem. Borůvka–Prim computes an MST.

Pf. Special case of the greedy algorithm.

Theorem. Borůvka–Prim can be implemented to run in $O(m \log \log n)$ time.

Pf.

- The $\log_2 \log_2 n$ phases of Borůvka's algorithm take $O(m \log \log n)$ time; resulting graph has $\leq n / \log_2 n$ nodes and $\leq m$ edges.
- Prim's algorithm (using Fibonacci heaps) takes $O(m + n)$ time on a graph with $n / \log_2 n$ nodes and m edges. ▪

$$O\left(m + \frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right)$$

Does a linear-time compare-based MST algorithm exist?

year	worst case	discovered by
1975	$O(m \log \log n)$	Yao
1976	$O(m \log \log n)$	Cheriton–Tarjan
1984	$O(m \log^* n)$, $O(m + n \log n)$	Fredman–Tarjan
1986	$O(m \log (\log^* n))$	Gabow–Galil–Spencer–Tarjan
1997	$O(m\, \alpha(n) \log \alpha(n))$	Chazelle
2000	$O(m\, \alpha(n))$	Chazelle
2002	*asymptotically optimal*	Pettie–Ramachandran
20xx	$O(m)$	??? 

deterministic compare-based MST algorithms

iterated logarithm function

$$\lg^* n = \begin{cases} 0 & \text{if } n \le 1 \\ 1 + \lg^*(\lg n) & \text{if } n > 1 \end{cases}$$

n	$\lg^* n$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 2^{16}]$	4
$(2^{16}, 2^{65536}]$	5

Theorem. [Fredman–Willard 1990] $O(m)$ in word RAM model.

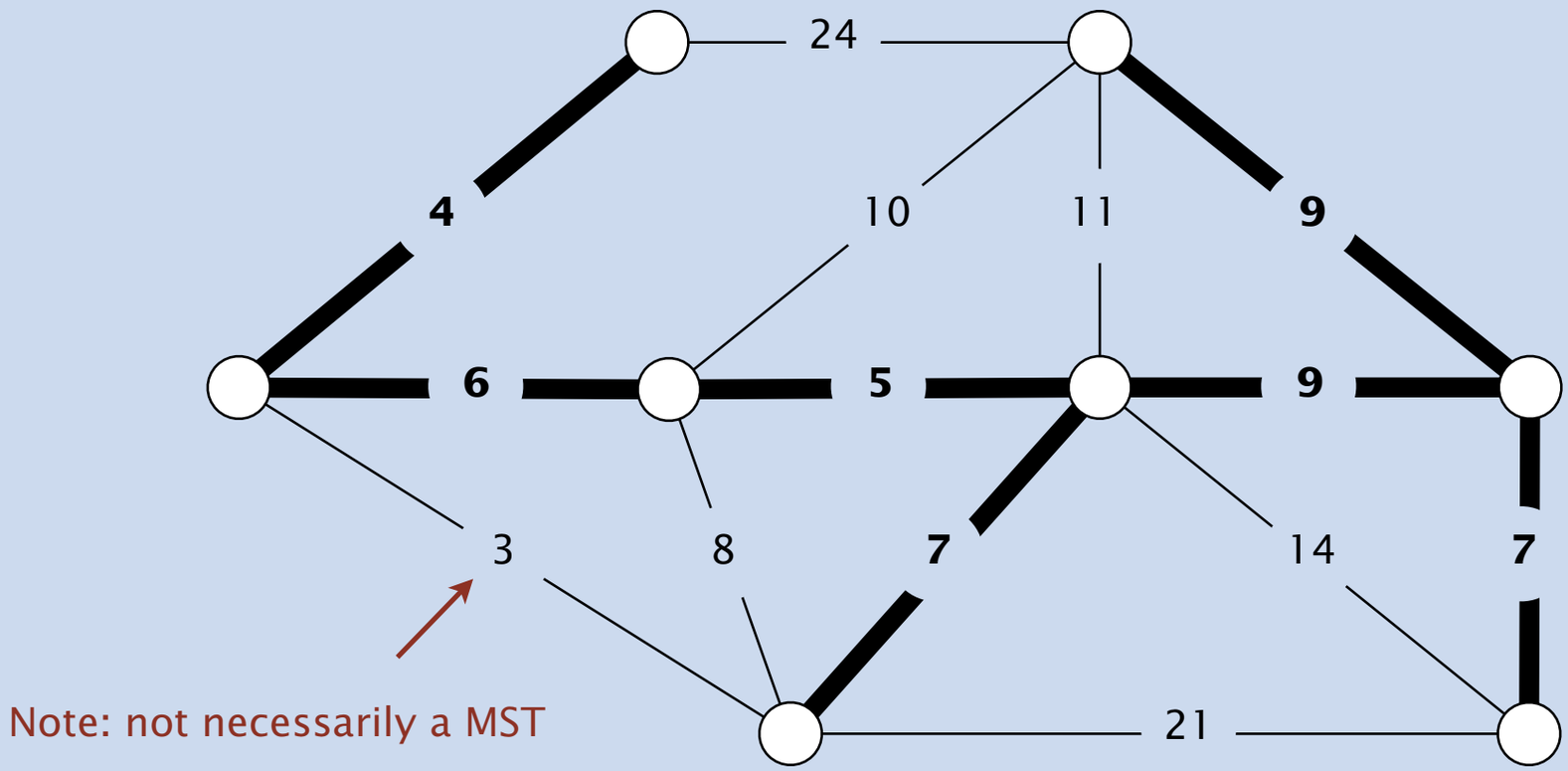
Theorem. [Dixon–Rauch–Tarjan 1992] $O(m)$ MST verification algorithm.

Theorem. [Karger–Klein–Tarjan 1995] $O(m)$ randomized MST algorithm.

Minimum Bottleneck Spanning Tree

Problem. Given a connected graph G with positive edge costs, find a spanning tree that minimizes the most expensive edge.

Goal. $O(m \log m)$ time or better.



minimum bottleneck spanning tree T (bottleneck = 9)

Greedy Algorithms

- Active selection (aka interval scheduling)
- Huffman coding
- Minimum spanning trees
- Take-home messages

Take-Home Messages

- Greedy algorithms solves optimization problems with specific sub-problem structures in comparison with dynamic programming.

- Even when the algorithm is actually sub-optimal, sometimes can lead to good approximation algorithms (discuss later in the course)

Take-Home Messages

- Greedy algorithms solves optimization problems with specific sub-problem structures in comparison with dynamic programming.

- Even when the algorithm is actually sub-optimal, sometimes can lead to good approximation algorithms (discuss later in the course)

Try to do more exercises.

Thanks for your attention!
Discussions?

Reference

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani: Chap. 5.

Algorithm design: Chap. 4.

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsII.pdf

Introduction to algorithms (4th edition): Chap. 15.

Data Structure and Algorithm Analysis in C (2nd Edition)： Sec. 10.3.

Book of Information Theory: https://www.inference.org.uk/itprnn/book.pdf